IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

# Runtime Hosting Interfaces

**Inventors:**

Weiwen Liu

Steven J. Pratschner

Ian H. Carmichael

Peter A. Carlin

Christopher W. Brumme

Mason K. Bendixen

Beysim Sezgin

Sean Trowbridge

Christopher James Brown

Mei-Chin Tsai

Mahesh Prakriya

Raja Krishnaswamy

Alan Chi-Hang Shi

Suzanne Maurine Cook

ATTORNEY'S DOCKET NO. MS1-1834US

## RELATED APPLICATIONS

[0001] This application is a continuation-in-part of U.S. Patent Application Serial No. 10/405,560, titled "Enhanced Runtime Hosting", filed on April 02, 2003, commonly assigned hereto, and hereby incorporated by reference.

## TECHNICAL FIELD

[0002] The invention pertains to integrated computer software application and hosted runtime service execution environments.

## BACKGROUND

[0003] In today's complex computer-program and networking environments, code sharing, scalability, and integration with other cross-platform frameworks are generally desired. Use of a runtime by a hosting application (i.e., hereinafter often referred to as a "host") generally allows application developers to write managed code with cross-platform compatibility, increased scalability, a common type system, multiple-language support, automatic memory management, and so on. Runtimes include, for example a Common Language Runtime (CLR), a Java Virtual Machine (VM), and/or the like.

[0004] Most hosts consist of both managed code and unmanaged code. Managed code is code that executes under the control of a runtime. Conversely, unmanaged code is code that runs outside of the runtime. Common object model (COM) components, ActiveX ® interfaces, and WIN32 ® API functions are examples of unmanaged code. Unmanaged hosting code is used by a process to configure a runtime, load it into the process, and transition the process into managed code.

[0005] Once the process, which includes the host and the runtime, is executing, integration between the host and the runtime in the execution environment is very limited and basic. Other than the described interactions of configuring, loading, and transitioning the runtime by the host, the host will typically only direct the runtime to perform a specific task, and/or receive an event/message from the runtime that a particular runtime task has completed. Thus, the host and the runtime are essentially separate, non-integrated entities of the process. This means that a hosting application has little or no control over many of the services provided by the runtime.

[0006] Relying on hosted runtime service(s) when the host expects use of its own service implementation can be problematic, and may even break the host application's expected performance, results, and/or the like. For instance, the host may have tuned specific threading, memory, synchronization, and/or security implementations over time, for example, for improved scalability and performance. Thus, although the host may desire the benefits of a runtime (e.g., cross-platform compatibility, reduced coding efforts, etc.), the host's specially tuned implementation may be incompatible with corresponding runtime services. In such a situation, the host may only load a runtime that relies on host-supplied services, or may completely bypass runtime services altogether by directly accessing underlying OS services. Such work-around(s) do not allow application designers to leverage the benefits that runtimes were designed to provide, resulting in less integrated and portable products.

[0007] Accordingly, systems and methods to increase execution environment control between a host and a runtime are greatly desired.

## SUMMARY

[0008] Systems and methods for enhanced runtime hosting are described. In one aspect the runtime hosting interface includes a host abstraction interface. The HAI allowing the runtime to configure host execution environment parameters and/or notify the host of a runtime event. In particular, the host abstraction interface (HAI) corresponds to execution environment abstractions supported by a host application. Responsive to an action or event, the runtime invokes an identified HAI or an associated object during execution of runtime managed code.

## BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The following detailed description is described with reference to the accompanying figures. In the figures, the left-most digit of a component reference number identifies the particular figure in which the component first appears.

[0010] Fig. 1 is a block diagram of an exemplary computing environment within which systems and methods for enhanced runtime hosting may be implemented.

[0011] Fig. 2 shows an exemplary procedure to provide enhanced runtime hosting.

[0012] Fig. 3 shows an exemplary architectural relationship between runtime and hosting application memory abstraction interfaces. For purposes of this discussion, an "abstraction" is a function/method that is optionally implemented by the application developer in the host application. The host-implemented function is abstract, because it will essentially replace a thread of execution that the runtime would have followed had the host not implemented the

function. In line with this, an application program interface used to access the host-implemented function is called an "abstracted" or "abstraction interface".

[0013] Fig. 4 shows an exemplary architectural relationship between runtime and hosting application thread abstraction interfaces.

[0014] Fig. 5 shows exemplary managed/unmanaged code call sequences for exiting/reverse-exiting and entering/reverse-entering managed code from/to unmanaged code to/from unmanaged code.

[0015] Fig. 6 is a block diagram showing exemplary task scheduling of runtime tasks that are treated as fibers scheduled on OS threads by a host application.

[0016] Fig. 7 shows an exemplary full stack of calls that were pushed and later popped between runtime and hosting application thread abstraction interfacing operations.

[0017] Fig. 8 shows how an exemplary implementation of the IHostAssemblyStore component fits into the assembly loading architecture of Fig. 1.

## DETAILED DESCRIPTION

### Overview

[0018] Systems and methods for enhanced runtime hosting are described. As discussed above, conventional integration between a runtime hosting application and the runtime is substantially limited in that the hosting application has very little if any control over a considerable portion of its execution environment. The invention addresses this lack of integration by providing substantially increased integration between the host and the runtime in the

execution environment, allowing host application to direct host-specific execution environment scenarios. In particular, this increased integration is implemented with multiple levels of abstracted interfaces that allow a host application to implement and exercise control and customize execution environment functionality, event notification, and runtime operation. Such abstractions include, for example, memory management, synchronization and threading, assembly loading, security context and impersonation, input/output (I/O) completion, event notification (e.g., when threads enter and leave the runtime), and runtime configuration.

[0019] Additionally, runtime interfaces provide the runtime with substantial control over specific host application implementation aspects, including event notification. Functionality of the runtime is completely independent on whether a particular hosting application has decided to implement hosting portions of the enhanced runtime interfaces.

## Exemplary Operating Environment

[0020] Turning to the drawings, wherein like reference numerals refer to like elements, the invention is illustrated as being implemented in a suitable computing environment. Although not required, the invention is described in the general context of computer-executable instructions, such as program modules, being executed by a personal computer. Program modules generally include routines, programs, objects, components, data structures, etc., that perform particular tasks or implement particular abstract data types.

[0021] Fig.1 illustrates an example of a suitable computing environment 100 on which the subsequently described systems, apparatuses and

methods for enhanced runtime hosting may be implemented (either fully or partially). Exemplary computing environment 100 is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of systems and methods the described herein. Neither should computing environment 100 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in computing environment 100.

[0022] The methods and systems described herein are operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well known computing systems, environments, and/or configurations that may be suitable for use include, but are not limited to, personal computers, server computers, multiprocessor systems, microprocessor-based systems, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and so on. Compact or subset versions of the framework may also be implemented in clients of limited resources, such as cellular phones, personal digital assistants, handheld computers, or other communication/computing devices. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

[0023] As shown in Fig. 1, computing environment 100 includes a general-purpose computing device in the form of a computer 102. The components of computer 102 can include, by are not limited to, one or more

processors or processing units 104, a system memory 106, and a bus 108 that couples various system components including system memory 106 to processor 104.

[0024] The system bus 108 represents one or more of any of several types of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnects (PCI) bus also known as Mezzanine bus.

[0025] Computer 102 typically includes a variety of computer readable media. Such media may be any available media that is accessible by computer 102, and it includes both volatile and non-volatile media, removable and non-removable media. In Fig. 1, system memory 106 includes computer readable media in the form of volatile memory, such as random access memory (RAM) 110, and/or non-volatile memory, such as read only memory (ROM) 112. A basic input/output system (BIOS) 114, containing the basic routines that help to transfer information between elements within computer 102, such as during start-up, is stored in ROM 112. RAM 110 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processor 104.

[0026] Computer 102 may further include other removable/non-removable, volatile/non-volatile computer storage media. For example, Fig. 1 illustrates a hard disk drive 116 for reading from and writing to a non-removable, non-volatile

magnetic media (not shown and typically called a "hard drive"), a magnetic disk drive 118 for reading from and writing to a removable, non-volatile magnetic disk 120 (e.g., a "floppy disk"), and an optical disk drive 122 for reading from or writing to a removable, non-volatile optical disk 124 such as a CD-ROM/R/RW, DVD-ROM/R/RW/+R/RAM or other optical media. Hard disk drive 116, magnetic disk drive 118 and optical disk drive 122 are each connected to bus 108 by one or more interfaces 126.

[0027] The drives and associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules, and other data for computer 102. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 120 and a removable optical disk 124, it should be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, random access memories (RAMs), read only memories (ROM), and the like, may also be used in the exemplary operating environment.

[0028] A user may provide commands and information into computer 102 through input devices such as keyboard 140 and pointing device 142 (such as a "mouse"). Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, serial port, scanner, camera, etc. These and other input devices are connected to the processing unit 104 through a user input interface 144 that is coupled to bus 108, but may be connected by other interface and bus structures, such as a parallel port, game port, or a universal serial bus (USB). A monitor 146 or other type of display device is also connected to bus 108 via an interface, such as a video adapter 148. In addition to monitor 146, personal

computers typically include other peripheral output devices (not shown), such as speakers and printers, which may be connected through output peripheral interface 150.

[0029] Computer 102 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 152. Remote computer 152 may include many or all of the elements and features described herein relative to computer 102. Logical connections shown in Fig. 1 are a local area network (LAN) 154 and a general wide area network (WAN) 156. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets, and the Internet.

[0030] When used in a LAN networking environment, computer 102 is connected to LAN 154 via network interface or adapter 158. When used in a WAN networking environment, the computer typically includes a modem 160 or other means for establishing communications over WAN 156. Modem 160, which may be internal or external, may be connected to system bus 108 via the user input interface 144 or other appropriate mechanism. Depicted in Fig. 1, is a specific implementation of a WAN via the Internet. Here, computer 102 employs modem 160 to establish communications with at least one remote computer 152 via the Internet 162.

[0031] In a networked environment, program modules depicted relative to computer 102, or portions thereof, may be stored in a remote memory storage device. Thus, e.g., as depicted in Fig. 1, remote application programs 164 may reside on a memory device of remote computer 152. It will be appreciated that the network connections shown and described are exemplary and other means of establishing a communications link between the computers may be used.

Exemplary Program Modules and Data for an Enhanced Runtime Interface

[0032] A number of program modules may be stored on the hard disk, magnetic disk 120, optical disk 124, ROM 112, or RAM 110, including, e.g., an operating system (OS) 128, a runtime 130, a host application 132 (hereinafter often referred to as "host" 132), other program modules 136, and program data 138. The OS provides functions, such as file management, event handling, processes and threads, memory management, user interfaces (e.g., windowing, menus, dialogs, etc.), security, authentication, verification, and/or the like.

[0033] The runtime 130 and the host application 132 expose runtime hosting interfaces 131 (RHIs) for application developers to abstract, customize, and tightly integrate process execution between the hosting application and the runtime. Exemplary such RHIs are presented in the Appendix beginning at page 26 of this document, wherein host interfaces 133 are prefaced with "IHost" and Runtime interfaces 134 are prefaced with "IRuntime." The RHIs substantially extend the functionality of conventional interfaces between runtimes and hosts by allowing host application(s) to customize and control many more aspects of the runtime 130 than possible in existing implementations. This provides substantially tighter execution environment integration between the runtime 130 and host 132 execution models as compared to conventional host/runtime execution environment integration.

[0034] For purposes of this discussion, an "abstraction" is a function/method that is optionally implemented by the application developer in the host application. The host-implemented function is abstract, because it will essentially replace a thread of execution that the runtime would have followed had

the host not implemented the function. In line with this, APIs exposed by the host 132 (i.e., host abstraction interfaces 133) for the runtime to redirect services to host-implemented functionality and/or exposed by the runtime 130 (i.e., runtime interfaces 134) for the host to notify the runtime of host action are called "abstracted" or "abstraction interfaces."

[0035] Host abstraction interfaces (HAIs) 133 are implemented by host application 132 and are mapped to any combination of memory, threading, I/O completion, synchronization, assembly loading, security context and impersonation, and/or other service functionalities provided by the host. Cooperative exchange between the runtime 130 and the host application 132 of information corresponding to the HAIs 133 allow the respective entities to negotiate which functions are to be performed by the host, and which functions are to be abstracted such that they are carried out by the runtime. As such the host application can customize its execution environment. Exposed HAIs further allow the runtime to configure certain host execution environment parameters, and notify the host (e.g., via supplied callbacks) of particular runtime events (e.g., resource allocation failures, thread state, etc.).

[0036] Runtime interfaces (RIs) 134 for use by the host application 132 to configure runtime operations, notify the runtime of certain events, to obtain additional information during process execution, and so on. During process execution, host application calls to the RI are: redirected back to the host application via one or more HAIs 133 for host specific implementation of a specific service, handed to the operating system 128 for execution, handled locally by the runtime, and/or communicated to object model services (i.e., "other program modules" 136).

## Memory Abstraction Interfaces

[0037] A memory management portion of the runtime hosting interfaces (RHIs) 131:

- Allow the host 132 to provide an interface (i.e., one of the exposed host application interfaces (HAIs) 133) through which the runtime 130 will request all memory allocations. In one implementation, the host interface 133 provides methods that replace both OS memory API's and standard C runtime allocation routines.

- Provide a mechanism for the host 132 to abstract the low memory notification the runtime 130 currently gets from the OS. This provides the host with a mechanism to ask the runtime 130 to make additional memory available, for example, via garbage collection services.

- Allow the runtime 130 to inform the host 132 of the consequences of failing a particular allocation, and further allow the host to customize the action the runtime should take if an allocation must be failed. For example, should the runtime unload an application domain or let it run in a "crippled" state.

[0038] The following scenario provides an exemplary use of the memory portion of the RHI 131. Suppose host 132 operates within a configurable amount of memory 110 (e.g., some or nearly all of the physical memory on the computing device 102). To maximize performance, the host tracks all memory allocations and insures that paging never occurs (the host of this example would rather fail a memory allocation than page to disk 116). To accurately track all allocations, the

host directs the runtime 130 to call an exposed host abstraction interface (HAI) 133 to allocate memory. This gives to the host the ability to fail the runtime's memory allocations before paging occurs.

## Threading Abstraction Interfaces

[0039] This section details the thread management portion of the RHI 131. These thread management hosting API's abstract the notion of an OS thread and essentially let the unit of scheduling and execution be defined by the host 132. This supports hosts that implement their own fiber-based scheduling mechanism. The term "task" is often used to define this abstraction and used to decouple the word "thread" from a particular host application's unit of execution and scheduling. In view of this, the thread management APIs:

- Allow the host 132 to provide an interface 133 that the runtime 130 will use to create and start new tasks (threads).

- Provide the host 132 with a mechanism to "reuse" or pool, a runtime-implemented portion of a task. This allows for performance optimization that can be used by a host application to minimize host-implemented task creation and setup operations.

- Implement a callback to notify the runtime 130 when a task has been moved to or from a runnable state. When a call is moved from a runnable state, the host API 133 allows the runtime 130 to specify that the task should be rescheduled by the host 132 as soon as possible.

- Provide a way for the runtime 130 to notify the host 132 that a given task cannot be moved to a different physical OS thread and cannot have its execution blocked during a specified window of time.

Allow the host 132 to provide an implementation of the thread pool, providing the runtime 130 with the ability to queue work items, set and query the size of the thread pool, and so on.

- Provide notifications on both the host 132 and runtime 130 side that the "locale" has been changed on a given task. A locale is related to localization of software. The runtime 130 includes a notion of current locale, and most hosts applications 132 do as well. These notification interfaces allow the runtime 130 and the host 132 to tell each other if the locale has been programmatically changed on either side – so both sides are kept in sync. For example, if the locale is changed on the runtime side, that may affect sorting order in a database implemented by the host 132.

- Allow the runtime 130 to delay host 132 abort of a given task.

- Provides means for the runtime 130 (and user code) to adjust the priority of a task.

[0040] The following scenario provides an exemplary use of the thread management portion of the RHI 131. Suppose that a particular host application 132 implements "fiber mode" execution. In fiber mode, a particular host (e.g., a scalable server side application in an SQL server) may create some number of threads based on the number of processors 104 on the computing device 102, or based on other host-specific criteria. The host then creates fibers on those threads on which to run user code (a portion of "other program modules" 136). The host schedules these fibers in a cooperative fashion (called non-preemptive in host terminology) - when a fiber blocks for some operation, it gets "switched out" and the thread runs another fiber. Later the fiber will get rescheduled and run—not necessarily on the same thread. When the runtime 130

creates a "task" through the hosting API 134, it ends up as a fiber in the host 132 and is natively understood by the host's scheduler.

## I/O Completion Abstraction Interfaces

[0041] This section details the I/O completion management portion of the RHI 131, wherein:

- The host 132 provides an HAI 133 for the runtime 130 to bind a handle to an I/O completion port.

- The host 132 provides an HAI 133 for the runtime 130 to supply a callback. The callback, for example, to be invoked by the host 132 when an asynchronous I/O operation completes.

- The runtime 130 provides an RI 134 for the host 132 to insert custom data at the end of the OVERLAPPED structure passed to runtime service I/O routines. An OVERLAPPED structure provides data-like file pointer position.

## Synchronization

[0042] If the runtime 130 creates one or more tasks through HAIs 133 (i.e., via direction of the host application 132), the runtime will also create the synchronization objects for those task(s) through corresponding HAIs as well. This ensures that locks are not taken on an OS thread without the host's knowledge. This allows runtime 130 tasks to further integrate with the hosts thread scheduling mechanism and to allow the host to perform deadlock detection. To this end, the synchronization management portion of the RHI 131 allows the runtime 130 to create the following synchronization ("sync") primitives through

the HAIs 133: critical sections, events (manual and auto-reset), semaphores, reader/writer locks, and monitors.

### Abstraction Interfaces for Entering and Leaving the Runtime

[0043] Threads running managed code can leave the runtime 130 to run unmanaged code. Locks taken on threads that leave the runtime 130 to run unmanaged code won't go through the RHI 131 so they can't be integrated with the threading and synchronization models of the host application 132. As such, the runtime notifies the host application via a host-implemented callback (the callback being provided by the host to the runtime through a corresponding RI 134) when a thread is entering or leaving the runtime 130 respectively to/from unmanaged code. In view if this, the runtime:

- Notifies the host 132 when a thread transitions into and out of the runtime 130. Such notifications are implemented by hooking calls out-of and into the runtime regardless of whether code has been compiled in a Just-In-Time (JIT) compiling scenario or in a native image compilation scenario (e.g., ngen). In one implementation, the notification includes the address of the routine being called.

- Allows the host 132 to specify that a particular function call to unmanaged code and corresponding runtime re-entry is not to be hooked by the runtime 130 for such notification. Such host specification allows the runtime to implement the particular call in an optimized manner (e.g., implementing the call inline). Hosts can use this interface to bypass the hook for calls they "know about" (i.e., calls that are either a part of the implementation of the host itself or of tightly integrated functionality).

[0044] For example, such a host-implemented callback allows the runtime 130 to send the host 132 a "notification" (a call to the hook) that tells the host that a particular thread's behavior can no longer be predicted since it has exited the runtime 130 into user code. Responsive to such a notification, the host 132 may take proactive steps to ensure that the particular thread is not scheduled by the host 132 to participate in any non-preemptive scheduling activity particular to the host's specific implementation, until the thread returns to the runtime.

[0045] In one implementation, such a hook can be used by the host 132 to adjust the floating point state of a processor 104. The host 132 may also utilize the RI 134 to indicate that one or more particular function calls are not to be hooked by the runtime 130, for example, to avoid runtime notifications when the host calls a data access API.

<u>Abstraction Interfaces for Assembly Loading</u>

[0046] The assembly loading abstraction, an exemplary implementation of which is described in the Appendix, section 1.10, comprises interfaces that allow hosts to customize the assembly loading process. Specifically, hosts can supply a list of assemblies that should be loaded domain-neutral and customize the way assemblies are located and loaded. In this implementation, interfaces in the Assembly Loading Abstraction include:

- **IHostAssemblyManager**. The runtime 130 asks for this top level interface through IHostControl::GetHostManager when the runtime 130 is initialized. If an implementation of this interface is provided, it is assumed that the host 132 wishes to control some aspect of the assembly binding process. IHostAssemblyManager contains methods for the host 132 to provide the list

of assemblies that should be loaded domain-neutral, the list of assemblies to which runtime 130 should bind, and to supply an implementation of IHostAssemblyStore through which the host 132 can implement their own custom assembly resolution process.

- **IHostAssemblyStore**. To load an assembly from somewhere other than the file system, a host 132 typically catches an AssemblyResolve event on System.AppDomain and provides a byte array containing the assembly's bits. An implementation of IHostAssemblyStore provides additional host-specific stores from which it can bind. If an IHostAssemblyStore is provided, runtime 130 will call back to the host 132 through this interface when binding to an assembly. The host 132 is free to load the assembly from anywhere it chooses and with whatever rules it deems appropriate. In essence, hosts can use IHostAssemblyStore to completely "bypass" Runtime 130 if so desired. A instance of the AssemblyBindInfo structure is passed to define the assembly each time a binding request occurs to the custom assembly store. This structure describes the identity of the assembly being requested along with information about whether there is any version policy present on the machine that might affect the bind.

- **IRuntimeAssemblyReferenceList**. Host 132 communicates information about assemblies, such as the list to load domain-neutral or the list loaded by the runtime 130 (not from the host 132 store) by creating a list of these assembly references, accessed by an instance of IRuntimeAssemblyReferenceList. This instance is created via IRuntimeAssemblyIdentityManager::GetRuntimeAssemblyReferenceList.

[0047] Each of these assembly loading abstraction interfaces are described in greater detail below in the Appendix, section 1.10

### Abstraction Interfaces for Security

[0048] Hosts 132 may choose to control all framework and user code access to thread tokens and to ensure complete security context information is passed across async points or points of restricted context execution. The actual context information unique to the host 132 is encapsulated in an instance of an interface IHostSecurityContext. This is opaque to the runtime 130 and will be captured and moved across threadpool worker item dispatch, finalizer execution, and both module and class constructor execution. Aspects of an exemplary security abstraction interface are described in greater detail below in the Appendix, section 1.11

### Abstraction Interfaces for Host Protection (1.12.4)

[0049] The .Net Framework class library provides an extensive set of built in functionality to hosted user code. In addition, third party class libraries exist to provide everything from statistical and math libraries to libraries of new UI controls. Yet, the full extent of functionality provided by a set of class libraries may not be appropriate in particular hosting scenarios. For example, displaying user interface in server programs or services is not useful, or allowing user code to exit the process cannot be allowed in hosts 132 that require long process lifetimes.

[0050] A host's designation of which categories of functionality to restrict can be thought of as falling into three general buckets:

- Categories of functionality that may cause the hosted user code to be instable, but do not affect the overall host 132 stability. These categories may not be

callable by partially-trusted code, but are ok to be called from fully-trusted code.

- Categories of functionality that do cause host 132 instability (i.e process exit). In this implementation, members in these categories are not be called in the host 132 process, regardless of the amount of trust the code has been granted.

- Categories of API's currently covered by Code Access permissions that a host 132 chooses to deny even in full-trust context. Hosts 132 restrict use of these API's via the *RefusedSet* property of the AppDomainManager class, which effectively injects a CAS deny at the tip of the thread's security evaluation.

[0051] To address this need to selectively configure certain hosting scenarios, IRuntimeHostProtectionManager provides the host 132 with a means to block classes, methods, properties and fields that offer a particular category of functionality from being loaded in the process. A host 132 may choose to prevent the loading of a class or the calling of a method for a number of reasons including reliability and scalability concerns, or because the functionality doesn't make sense in that host's environment as in the examples described above.

[0052] These and other aspects of an exemplary host protection abstraction interface are described in greater detail below in the Appendix, section 1.12.4.

## Abstraction Interfaces for Debugging (1.12.2)

[0053] For debugging purposes, hosts 132 may want to group tasks by some host-specific logical construct like a connection, session or request. In this way, a developer who is debugging a particular session (for example) only sees the tasks

involved in processing that session – it does not see every task running in the process. This interface provides methods that allow hosts 132 to associate a list of runtime tasks with a given id and friendly name. In this implementation, the id, the name, and the list of associated tasks have meaning independent of the runtime 130 (for purposes of this call). That is, the runtime 130 blindly passes the parameters on to the debugger.

[0054] Get and SetDacl methods allow the host 132 to either set or retrieve the Access Control Lists (ACLs) on events and shared memory used by the debugger. If the events or shared memory are already in use, setting new ACL's will fail. Likewise, upon creation, restrictive ACL's can be set which disallow debugging (access-denied ACE's in the ACL). If A caller wants to preserve the semantic of the default ACL values when calling SetDacl, the Administrators group and the current process owner may be added to the ACL, in addition to other required users. If GetDacl is called before SetDacl, the returned ACL is the default ACL placed on the debug memory block by the runtime 130.

[0055] These and other aspects of an exemplary debugging abstraction interface are described in greater detail below in the Appendix, section 1.12.2.

## Abstraction Interfaces for Shutting Down the Runtime

[0056] During process shutdown, an operating system (OS) may terminate threads at unpredictable points, possibly while they still hold locks taken through a host's critical section implementation. In traditional implementations, those locks are orphaned and future threads which attempt to acquire them will block. If the remaining live thread, executing under the OS loader lock, attempts to acquire one of these orphaned locks, the process will deadlock.

[0057] To avoid this, the systems and methods for enhanced runtime hosting described herein allow the host application 132 (Fig. 1) to imitate the OS's process shutdown behavior. When a process is shutting down, the OS will allow locks orphaned by other threads to be entered by the current thread holding the OS loader lock. This effectively makes locks irrelevant during shutdown because any lock can be successfully taken.

[0058] In one implementation, there are several ways the host 132 can be aware of process shutdown. These implementations describe APIs that are discussed in greater detail below. The first is that the host calls ExitProcess. Here the host has an opportunity to notify its locks, for example by setting a flag globally available to all locks, that they should feel free to let threads enter and leave freely. The host can also use the IActionOnCLREvent notification. The host will receive notification of the CLR becoming disabled and at this point a host can again inform its locks that they should let threads enter/leave freely.

[0059] Additionally, the host 132 may use another tactic. The host can prohibit managed code from calling ExitProcess by denying the code access permission required. To exit, the host can then call TerminateProcess and DllMain will not to run at all. This ensures that no deadlocks occur.

## An Exemplary Procedure for Enhanced Runtime Hosting

[0060] Fig. 2 shows an exemplary procedure for enhanced runtime hosting. For purposes of discussion, the operations of Fig. 2 are discussed in reference to features of Fig. 1. Although Fig. 2 shows operations 204-212 in a particular numerical order, these respective operations could logically occur in any order as a function of host application 132 and runtime 130 interaction, characteristics of the execution environment, and so on.

[0061] At block 202, after the host application 132 (Fig. 1) has loaded the runtime 130 (Fig. 1) into a process (i.e., startup time), the runtime 130 determines any host-implemented execution environment abstractions (i.e., host supported abstractions). For purposes of discussion, the process is represented by the respective host 132 and runtime 130 components in RAM 110 (Fig. 1). (Whether or not a particular abstraction is supported by the host is not central to how the runtime executes). In one implementation, this operation is accomplished with a request/call by the runtime to an exposed HAI 133 with a list indicating execution environment abstractions of interest. Such a request is represented as a portion of "program data" 138 of Fig. 1. Responsive to this request/call, the host provides the runtime 130 with one or more abstraction-specific interfaces (respective APIs in the HAIs 133) through which to access host-implemented execution environment abstractions (e.g., memory, thread, synchronization, I/O completion, policy, and/or other implemented abstractions).

[0062] At block 204, responsive to the runtime request for information regarding which, if any, execution environment abstractions are implemented/supported by the host application 132, the runtime 130 receives a list

of HAIs 133 that correspond to abstracted functionality. Such HAIs may reference objects and/or interfaces. At block 206, the runtime may configure abstracted functionality implementation(s) via one or more of the received HAIs.

[0063] At block 208, the runtime 130 notifies the host application 132 of one or more runtime interfaces 134 (RIs) exposed by the runtime. In this implementation, such a notification is responsive to a request sent to the runtime 130 by the host application 132. Such a notification and request are represented as respective portions of "program data" 138 of Fig. 1. Such RIs allow the host to notify the runtime of certain events and/or to obtain additional information during process execution. In one implementation, the RI is a callback utilized by the host to notify the runtime of certain events or to obtain additional information during process execution. At block 210, the runtime responds to any host application configuration requests, request for a callback, runtime data, and/or the like.

[0064] At block 212, during execution of managed code and responsive to one or more actions or events associated with host-application abstracted functionality, the runtime calls at least one specific interface or object corresponding to a specific one of the returned HAIs. Such actions and events correspond, for example, to management services for memory, threads/tasks, I/O completion, synchronization, event notification(s), garbage collection (GC), and/or the like.

## Conclusion

[0065] The described systems and methods provide enhanced runtime hosting. Although the systems and methods have been described in language specific to structural features and methodological operations, the subject matter as defined in the appended claims are not necessarily limited to the specific features or operations described. Rather, the specific features and operations are disclosed as exemplary forms of implementing the claimed subject matter.

# APPENDIX
# EXEMPLARY RUNTIME HOSTING INTERFACES

1      This APPENDIX to the detailed description is described with reference to

2   the accompanying figures.  In the figures, the left-most digit of a reference number

3   identifies the particular figure in which the component first appears.

4

5                              **CONTENTS**

**1. Functional Design**

1

2          Several new interfaces are provided to support the abstractions described

3 above. A host 132 implements a given abstraction by providing the runtime 130

4 with an abstraction-specific interface through which to call. For example, all

5 memory allocations made by the runtime 130 will go through the host 132 if the

6 host 132 provides an implementation of the IHostMemoryManager interface.

7

8          At startup time, the runtime 130 determines which abstractions the host 132

9 implements using the IRuntimeRuntimeHost and IHostControl interfaces. An

10 IRuntimeRuntimeHost interface pointer is obtained by calling a method used to

11 initialize the runtime 130. IRuntimeRuntimeHost provides the same methods as

12 ICorRuntimeHost did in V1 (minus the obsolete ones), plus one additional

13 SetHostControl. The host 132 calls SetHostControl and provides a pointer to its

14 implementation of IHostControl.

15

16          The runtime 130 then calls the GetHostManager method on IHostControl

17 with IID's corresponding to the abstraction-specific objects and interfaces to see

18 which abstractions the host 132 supports. For example, to determine if the host

19 132 supports the memory abstraction, the runtime 130 will call

20 IHostControl::GetHostManager with the IID for IHostMemoryManager and host

21 132 memory manager. If an interface is returned, the runtime 130 will call

22 through that interface for all memory allocations made during the lifetime of the

23 process.

1    In some cases, the abstraction-specific interfaces also contain callbacks that

2    allow the host 132 to notify the runtime 130 of certain events or to obtain

3    additional information at runtime.

4

5    1.1. Naming Conventions

6    The names of the interfaces implemented by the host 132 start with "IHost"

7    while the names of the callbacks implemented by the runtime 130 start with

8    "IRuntime".

9

10   1.2. Common HRESULTs

11   The following HRESULTs may be returned from any of the API's

12   described in this document:

13   • **S_OK**. Success.

14   • **HOST_E_RuntimeNOTAVAILABLE**. The runtime 130 has not been

15       loaded into a process or is in a state in which it cannot run managed code or

16       successfully process the call.

17   • **E_FAIL**. A method will only return *E_FAIL* when an unknown,

18       catastrophic failure has occurred. After any method returns *E_FAIL*, the

19       runtime 130 is no longer usable within the process. Subsequent calls to any

20       Hosting API will return *HOST_E_RUNTIMENOTAVAILABLE*

21

22   If a given method returns other HRESULTs, those other results will be

23   listed in the documentation for that method.

1    1.3. Deferred Runtime Startup

2        Hosts 132 may wish to determine the version of the runtime which will be

3    used within the process before explicitly initializing the runtime.  An API is

4    provided for this purpose.

5

```
6    typedef HRESULT (__stdcall *FLockClrVersionCallback) ();
7    STDAPI LockClrVersion(FLockClrVersionCallback hostCallback,
8                FLockClrVersionCallback *pBeginHostSetup,
9                FLockClrVersionCallback *pEndHostSetup).
```

10       LockClrVersion is called by the host 132 prior to runtime initialization.

11   When the runtime is first initialized, either through a call to one of the

12   CorBindToRuntime* functions, or by COM object activation, the following

13   occurs:

14   •    hostCallback is called by the runtime

15   •    From within the callback, the host 132 calls:

16        o  *pBeginHostSetup

17        o  CorBindToRuntime*

18        o  IRuntimeRuntimeHost::SetHostControl

19        o  IRuntimeRuntimeHost::Start

20        o  *pEndHostSetup.

21

22       Everything from *pBeginHostSetup until *pEndHostSetup happens on one

23   thread/fiber with the same logical stack, which may be different from the thread

24   from which hostCallback is called.

25

1    **1.4. Top-Level Interfaces**

2    **1.4.1. IRuntimeRuntimeHost**

3        This interface provides the functionality of the ICorRuntimeHost interface

4    implemented in V1, plus the additional method to set the host 132 control

5    interface.  Hosts 132 get a pointer to an IRuntimeRuntimeHost by calling one of

6    the APIs used to bind or initialize the runtime 103.  All hosts 132 that wish to

7    provide implementations of one of more of the host 132ing abstractions may use

8    this interface instead of ICorRuntimeHost.

9

```
10    interface IRuntimeRuntimeHost : IUnknown
11    {
12      HRESULT Start();
13
14      HRESULT Stop();
15
16      HRESULT CreateDomain([in] LPCWSTR pwzFriendlyName,
17                  [in] IUnknown* pIdentityArray, // Optional
18                  [out] IUnknown** pAppDomain);
19
20      HRESULT GetDefaultDomain([out] IUnknown** pAppDomain);
21
22      HRESULT EnumDomains([out] HDOMAINENUM *hEnum);
23
24      HRESULT NextDomain([in] HDOMAINENUM hEnum,
25                  [out] IUnknown** ppppDomain);
26
27      HRESULT CloseEnum([in] HDOMAINENUM hEnum);
28
29      HRESULT CreateDomainEx([in] LPCWSTR pwzFriendlyName, // Optional
30                  [in] IUnknown* pSetup,      // Optional
31                  [in] IUnknown* pEvidence,    // Optional
32                  [out] IUnknown** pAppDomain);
33
34      HRESULT CreateDomainSetup([out] IUnknown** pAppDomainSetup);
35
36      HRESULT CurrentDomain([out] IUnknown** pAppDomain);
37
```

1       HRESULT SetHostControl([in] IHostControl* pHostControl);

2

3       HRESULT GetRuntimeControl([out] IRuntimeControl** pRuntimeControl);

4

5       HRESULT UnloadAppDomain([in] DWORD dwAppDomainId);

6       };

7

8       *1.4.1.1.      Start*

9       Initializes the runtime 130 into the process.  After this call, the runtime 130

10      is ready to run managed code.  Note in many scenarios it is not necessary to call

11      Start explicitly as the runtime 130 will initialize itself on the first request to run

12      managed code.  However, calling Start is convenient for hosts that want to control

13      exactly when the runtime 130 initializes.

14

15      **HResults**

16      **See Common HResults**

17

18      *1.4.1.2.      UnloadDomain*

19      Unloads the specified AppDomain from the process.  Thread.Abort is

20      called on each thread running in the domain.  When all threads have terminated

21      the domain is unloaded.

22

| Parameter | Description |
|---|---|
| pAppDomain | [in] A pointer of type _AppDomain to the domain you want to unload. |

1

2 **HResults**

3 See **Common HResults**

4 **COR_E_CANNOTUNLOADAPPDOMAIN**. The specified AppDomain

5 could not be aborted. The most common cause of this error is a thread running in

6 the domain could not be terminated. Other reasons include an attempt to unload

7 the default domain or an attempt to unload a domain that has already been

8 unloaded.

9

10 *1.4.1.3. CurrentDomain*

11 Returns an interface pointer to the AppDomain that is running on the

12 calling thread.

13

| Parameter | Description |
|-----------|-------------|
| pAppDomain | [out] A pointer of type _AppDomain representing the calling thread's domain. |

14

15 **HResults**

16 **See Common HResults**

17

18 *1.4.1.4. SetHostControl*

19 SetHostControl takes an interface pointer to the host 132's implementation

20 of IHostControl. This method is be called before the runtime has been initialized,

1    i.e before calling IRuntimeRuntimeHost::Start or before any metadata interfaces

2    are used.  It is recommended that this method be called immediately after the

3    return from CorBindToRuntime*.

4

| Parameter | Description |
|---|---|
| pHostControl | [in]    The    host    132's implementation of IHostControl. |

5

6        **HResults**

7        **See Common HResults**

8        **E_RUNTIME_ALREADY_STARTED**. The runtime 130 has already been

9        initialized.  The call to SetHostControl did not have any affect.

10

11    *1.4.1.5.        GetRuntimeControl*

12        GetRuntimeControl returns an interface pointer of type IRuntimeControl

13    that hosts can use to those runtime 130 "managers" that have no corresponding

14    interface on the host 132 side.  These managers are used to customize additional

15    aspects of the runtime.  See Runtime Configuration Interfaces for details.

16

| Parameter | Description |
|---|---|
| pRuntimeControl | [out] An interface pointer that hosts can use to set additional configuration parameters for the |

| | runtime 130. |
|---|---|

1

2    **HResults**

3    **See <u>Common HResults</u>**

4

5    *1.4.1.6.        UnloadAppDomain*

6    Unmanaged analog to the AppDomain.Unload method.  Allows unmanaged

7    hosts to unload a specific domain by its numeric ID.

8

| Parameter | Description |
|---|---|
| dwAppDomainId | [in] numeric ID of Application Domain to unload. |

9

10   *1.4.1.7.      ExecuteInDomain*

11   Allows the host 132 to control which appdomain will execute selected

12   managed code.

13

14   HRESULT ExecuteInDomain([in] DWORD AppDomainId, [in]
15   FExecuteInDomainCallback pCallback, [in] void* cookie);

16

| Parameter | Description |
|---|---|
| dwAppDomainId | [in] index of AppDomain in |

|  | which to execute |
|---|---|
| FExecuteInDomainCallback | [in] function pointer to execute in selected appdomain |
| cookie | [in] pointer to opaque caller-allocated memory. This parameter is passed by the runtime 130 to the domain callback. It is unaltered by the runtime 130 and both allocation and lifetime of this memory are controlled by the caller. This is not runtime 130 managed heap memory. |

1

2    **1.4.2. IHostControl**

3        This is the interface the runtime 130 uses to determine which abstractions

4    the host 132 supports.

5

6    interface IHostControl : IUnknown

7    {

8        HRESULT GetHostManager(

9            [in] REFIID riid,

10            [out, iid_is(riid)] IUnknown** ppObject);

11

12        HRESULT GetAppDomainManagerType(

13            [in, out] DWORD *pcchAssemblyName,

14            [out, size_is(*pcchAssemblyName)] wchar_t *pchAssemblyName,

15            [in, out] DWORD *pcchTypeName,

16            [out, size_is(*pcchTypeName)] wchar_t *pchTypeName);

17

18        HRESULT SetAppDomainManager(

1        [in] DWORD dwAppDomainID,

2        [in] IUnknown* pUnkAppDomainManager);

3    }

### 4   *1.4.2.1.    GetHostManager*

5        GetHostManager is called by the runtime 130 to determine if the host 132

6   supports a given abstraction interface.  The specific interfaces a host 132 will be

7   asked for are:

8      • IID_IHostMemoryManager

9      • IID_IHostTaskManager

10     • IID_IHostThreadPoolManager

11     • IID_IHostIoCompletionManager

12     • IID_IHostSyncManager

13     • IID_IHostAssemblyManager

14     • IID_IHostCrossAssemblyCallManager

15     • IID_IHostGCManager

16

| Parameter | Description |
|---|---|
| riid | [in] The IID of the abstraction interface the runtime 130 is asking for. i.e **IID_IHostMemoryManager** |
| ppObject | [out] A pointer to the host 132 implemented abstraction interface or NULL if the host 132 doesn't support the |

| | |
|---|---|
| . | abstraction. |

1

2    **HResults**

3    **See <u>Common HResults</u>**

4    **E_INVALID_ARG.**  The requested CLSID is not valid.

5    **E_NOINTERFACE.**  The requested interface is not supported.

6

7    *1.4.2.2.        GetAppDomainManagerType*

8    This method allows the host 132 to pass information for a managed type

9    derived from AppDomainManager.  The host 132 passes the name of the assembly

10   implementing the type, and the type name.  The runtime 130 loads the assembly

11   and instantiates an object of this type.  This object will be notified at creation of

12   each AppDomain.

13

| Parameter | Description |
|---|---|
| Parameter | Description |
| pcchAssemblyName | [in,out] size in characters for the assembly name buffer (next parameter) |
| pchAssemblyName | [out] buffer which contains assembly name at return of call |
| pcchTypeName | [in,out] size in characters for the type name buffer (next parameter) |
| pchTypeName | [out] buffer which contains type |

| | name at return of call |
|---|---|

1

2    In this implementation, this call is made twice. First, to ascertain the sizes

3    needed for the buffers, the runtime 130 passes NULL for pchAssemblyName and

4    pchTypeName. Upon return, pcchAssemblyName and pcchTypeName should

5    contain the necessary sizes for the buffers. On the second call,

6    pcchAssemblyName and pcchTypeName will be set to these sizes with

7    pchAsssemblyName and pchTypeName pointing at buffers allocated by the

8    runtime 130. The host 132 should then fill in the buffers with appropriate

9    information.

10

11    *1.4.2.3.    SetAppDomainManager*

12

| Parameter | Description |
|---|---|
| Parameter | Description |
| dwAppDomainID | [in] numeric ID of the chosen AppDomain |
| pUnkAppDomainManager | [in] A pointer to the host 132 implemented AppDomainManager object (as IUnknown) |

13

14    1.5. Memory Abstraction Interfaces

15    There are three interfaces in the memory abstraction:

1       **IHostMemoryManager.** Implemented by the host 132 and discovered by

2       the runtime 130 via IHostControl::GetHostManager. Allows the runtime

3       130 to make virtual memory requests through the host 132 instead of

4       calling the standard Win32 virtual memory API's. In addition, this

5       interface provides methods for the runtime 130 to obtain a pointer through

6       which to make memory requests on the heap and to get back the host 132's

7       reported level of memory pressure in the process.

8     •      **IHostMalloc.** Implemented by the host 132. Allows the runtime 130 to

9       make fine grained allocations from the heap through the host 132.

10    •      **IRuntimeMemoryNotificationCallback.** Implemented by the runtime

11       130. The host 132 invokes this callback to request memory be freed.

12

13       Fig. 4 shows an exemplary architectural relationship between runtime and

14 hosting application thread abstraction interfaces

15

16 **1.5.1. IHostMemoryManager**

17

18      typedef enum

19      {

20       eMemoryAvailableLow = 1,

21        eMemoryAvailableNeutral = 2,

22        eMemoryAvailableHigh = 3

23      } EMemoryAvailable;

24

25      typedef enum

26      {

27       eTaskCritical = 0,

28       eAppDomainCritical = 1,

29       eProcessCritical = 2

30      } EMemoryCriticalLevel;

31

```
1      interface IHostMemoryManager : IUnknown
2      {
3        HRESULT CreateMalloc([in] BOOL fThreadSafe,
4                             [out] IHostMalloc **ppMalloc);
5
6        HRESULT VirtualAlloc([in] void*      pAddress,
7                          [in] SIZE_T     dwSize,
8                          [in] DWORD       flAllocationType,
9                          [in] DWORD       flProtect,
10                     [in] EMemoryCriticalLevel dwCriticalLevel,
11                     [out] void**     ppMem);
12
13       HRESULT VirtualFree([in] LPVOID     lpAddress,
14                         [in] SIZE_T     dwSize,
15                         [in] DWORD       dwFreeType);
16
17       HRESULT VirtualQuery([in] void *    lpAddress,
18                         [out] void*     lpBuffer,
19                         [in] SIZE_T     dwLength,
20                     [out] SIZE_T * pResult);
21
22       HRESULT VirtualProtect([in] void *      lpAddress,
23                         [in] SIZE_T      dwSize,
24                         [in] DWORD       flNewProtect,
25                         [out] DWORD *    pflOldProtect);
26
27       HRESULT GetMemoryLoad([out] DWORD* pMemoryLoad,
28                         [out] SIZE_T *pAvailableBytes);
29
30     HRESULT RegisterMemoryNotificationCallback([in]
31     IRuntimeMemoryNotificationCallback
32
33       * pCallback);
34     }
```

### 1.5.1.1.      Enum: EMemoryAvailable

This enumeration contains values that indicate the amount of free physical

memory on the machine.  These values logically map to the events for high and

low memory returned from Win32's CreateMemoryResourceNotification.

- **eMemoryAvailableLow**:  Available physical memory is running low.

1 • **eMemoryAvailableNeutral**: Available physical memory is neutral.

2 **eMemoryAvailableHigh**: Available physical memory is high.

3 This value is passed into the runtime 130 from the host 132 through the

4 "memory notification" callback described below.

5

6 *1.5.1.2.       Enum: EMemoryCriticalLevel*

7 The values in this enumeration indicate the impact of failing a particular

8 memory allocation.  The memory allocation routines in IHostMemoryManager

9 and IHostMalloc take a parameter of this type.  Based on the severity of failing the

10 request, a host 132 can decide to fail the allocation request immediately or wait

11 until it can be satisfied.

12

13 Exemplary values in the enumeration are:

14 • **TaskCritical**: This allocation is critical to running the task from which the

15 allocation is made.  If memory cannot be allocated, the runtime 130 cannot

16 guarantee that the task is in a consistent, runnable state.  As such, the

17 runtime 130 will raise a thread abort exception on the physical OS thread.

18 • **AppDomainCritical**: This allocation is critical to continue executing

19 managed code in the domain requesting the allocation.  If memory cannot

20 be allocated, the runtime 130 cannot guarantee that the domain is still

21 usable. The decision about what action should be taken when the allocation

22 cannot be satisfied is left to the host 132.  The host 132 can tell the runtime

23 130 to automatically abort the AppDomain or allow it to keep running by

24 calling methods on IRuntimePolicyManager.

1       **ProcessCritical**: This allocation is critical to continue executing managed

2       code in the process. If memory cannot be allocated, the runtime 130 cannot

3       function in the process. ProcessCritical is used during startup and when

4       running finalizers. If allocation cannot be satisfied, the runtime 130 is

5       essentially disabled – it cannot be used to run any managed code in the

6       process. All subsequent calls into the runtime 130 will fail with

7       HOST_E_RUNTIMENOTAVAILABLE.

8

9       *1.5.1.3.     CreateMalloc*

10      Returns a pointer to an IHostMalloc used to make allocations from a host-

11   created heap (see IMalloc below). In this implementation, true is passed for

12   fThreadSafe as a function of performance numbers.

13

| Parameter | Description |
|---|---|
| fThreadSafe | [in] If true, the memory returned can me accessed by multiple threads without any synchronization. If false, calls on the object should be serialized. |
| ppMalloc | [out] A pointer to a host-implemented IHostMalloc. |

14      **HResults**

15      **See <u>Common HResults</u>**

16      **E_OUTOFMEMORY**. Memory not available to complete this request.

17

1    *1.5.1.4.        VirtualAlloc*

2    This method logically wraps the Win32 function of the same name.

3

| Parameter | Description |
|---|---|
| pAddress | [in] As defined in Win32's VirtualAlloc. |
| dwSize | [in] As defined in Win32's VirtualAlloc. |
| flAllocationType | [in] As defined in Win32's VirtualAlloc. |
| flProtect | [in] As defined in Win32's VirtualAlloc. |
| dwCriticalLevel | [in] A value indicating the impact of failing the allocation. See description of EMemoryCriticalLevel above. |
| ppMem | [out] Pointer to the allocated memory or NULL if the request couldn't be satisfied. Note that Win32's VirtualAlloc returns this pointer as the return value from the function. |

4

5    **HResults**

6    **See Common HResults**

1      **E_OUTOFMEMORY**. Memory not available to complete this request.

2

3     *1.5.1.5.*       *VirtualFree*

4      This method logically wraps the Win32 function of the same name.

5

| Parameter | Description |
|---|---|
| lpAddress | [in] As defined in Win32's VirtualFree. |
| dwSize | [in] As defined in Win32's VirtualFree. |
| dwFreeType | [in] As defined in Win32's VirtualFree. |

6

7      **HResults**

8     **See <u>Common HResults</u>**

9        **HOST_E_INVALIDOPERATION.** An attempt was made to free

10    memory not allocated through the host 132.

11

12     *1.5.1.6.*       *VirtualQuery*

13      This method wraps the Win32 function of the same name. Note that the

14 return type is passed back through the pResult out parameter instead of as the

15 function's return value as done in Win32. VirtualQuery, when implemented by the

16 OS, does not incur deadlock and may run to completion with random threads

17 suspended in user code. However, when implemented by a host, care should be

1 taken not to take locks in the implementation of this method to avoid deadlock

2 between threads executing in user space to best approximate OS behavior.

3

4 **HResults**

5 See **Common HResults**

6

7 *1.5.1.7.        VirtualProtect*

8 This method wraps the Win32 function of the same name. All parameters

9 have the same semantics as the corresponding Win32 function.

10

11 **HResults**

12 See **Common HResults**

13

14 *1.5.1.8.        GetMemoryLoad*

15 This method is called by the runtime 130 to get the amount of physical

16 memory that is in use (i.e not free) as reported by the host 132. The runtime 130

17 uses this value as a heuristic to the GC. For example, if the host 132 reports that

18 the majority of memory is in use the GC may elect to collect from multiple

19 generations to increase the amount of memory that may potentially become

20 available.

21

22 This method wraps Win32's GlobalMemoryStatus. The value returned

23 from GetMemoryLoad in pMemoryLoad is the equivalent of the dwMemoryLoad

1     field in the MEMORYSTATUS structure returned from GlobalMemoryStatus.

2     pAvailableBytes returns the number of bytes available to the runtime.

3

| Parameter | Description |
|---|---|
| pMemoryLoad | [out] The approximate percentage of total physical memory that is currently in use. |
| pAvailableBytes | [out] The number of bytes available to the runtime. |

4

5     **HResults**

6     **See Common HResults**

7

8     *1.5.1.9.*     *RegisterMemoryNotificationCallback*

9     The "memory notification callback" is passed into the host 132 by the

10     runtime 130. The host 132 invokes this callback to notify the runtime 130 of the

11     current memory load on the machine. See complete description of

12     IRuntimeMemoryNotificationCallback below.

13

| Parameter | Description |
|---|---|
| pCallback | [in] An interface pointer to a runtime 130-provided implementation of IRuntimeMemoryNotificationCallback. |

14

1     **HResults**

2     **See <u>Common HResults</u>**

3

4     **1.5.2. IHostMalloc**

5     This interface allows the runtime 130 to make fine grained allocations from

6  the heap through the host 132.

7

8     interface IHostMalloc : IUnknown
9     {
10        HRESULT Alloc([in] SIZE_T  cbSize,
11             [in] EMemoryCriticalLevel dwCriticalLevel,
12               [out] void** ppMem);
13
14        HRESULT DebugAlloc([in] SIZE_T    cbSize,
15             [in] EMemoryCriticalLevel dwCriticalLevel,
16               [in] char*      pszFileName,
17               [in] int        iLineNo,
18               [out] void**    ppMem);
19
20        HRESULT Free([in] void* pMem);
21     }

22

23   *1.5.2.1.    Alloc*

24     Allocates the requested amount of memory from the heap.

25

| Parameter | Description |
|---|---|
| cbSize | [in] The amount of memory being requested in bytes. |
| dwCriticalLevel | [in] A value indicating the impact of failing the allocation. See |

| | |
|---|---|
| | description of EMemoryCriticalLevel above. |
| ppMem | [out] Pointer to the allocated memory or NULL if the request could not be satisfied. |

1    **HResults**

2    **See Common HResults**

3    **E_OUTOFMEMORY**. Memory not available to complete this request.

4

5    *1.5.2.2.    DebugAlloc*

6    The debug version of Alloc. DebugAlloc tracks where the memory was

7    allocated.

8

| Parameter | Description |
|---|---|
| cbSize | [in] The amount of memory being requested in bytes. |
| dwCriticalLevel | [in] A value indicating the impact of failing the allocation. See description of EMemoryCriticalLevel above. |
| pszFileName | [in] file name |
| iLineNo | [in] The line number within pszFileName where the allocation occurred. |

| | |
|---|---|
| ppMem | [out]  Pointer to the allocated memory or NULL if the request could not be satisfied. |

1

2      **HResults**

3      **See Common HResults**

4      **E_OUTOFMEMORY**. Memory not available to complete this request.

5   *1.5.2.3.      Free*

6      Frees memory allocated with Alloc.

7

| Parameter | Description |
|---|---|
| pMem | [in] A pointer to the memory to be freed. |

8

9      **HResults**

10     **See Common HResults**

11     **HOST_E_INVALIDOPERATION.** An attempt was made to free memory

12     not allocated through the host 132.

13

14   **1.5.3. IRuntimeMemoryNotificationCallback**

15     The runtime 130 passes this callback to the host 132 through

16   IHostMemoryManager:RegisterRuntimeMemoryNotificationCallback.      The

17   purpose of this interface is to allow the host 132 to report memory pressure

18   conditions similar to the Win32 API CreateMemoryResourceNotification.

1

2  interface IRuntimeMemoryNotificationCallback : IUnknown
3  {
4      HRESULT OnMemoryNotification([in] EMemoryAvailable
5  eMemoryAvailable);
6  }

7

8  *1.5.3.1.        OnMemoryNotification*

9       OnMemoryNotification notifies the runtime 130 of the memory load on the

10  machine.  The runtime 130 will use this information to free up additional memory

11  when the host 132 reports that the amount of available physical memory is low.

12  Calls to OnMemoryNotifcation always return immediately – non-blocking.

13

| Parameter | Description |
|---|---|
| eMemoryAvailable | [in]      A     value     from EMemoryAvailable indicating whether the amount of free physical memory on the machine is low or high. |

14

15       **HResults**

16       **See Common HResults**

17

18  1.6. Threading Abstraction Interfaces

19       As described, hosting API's use the term "task" to abstract the notion of

20  what would typically be thought of as a thread.  This is done to decouple the term

1   "thread" from a given hosts unit of execution and scheduling. An example of why

2   this separation is used to support ha host with "fiber mode" execution model. In

3   fiber mode, tasks become fibers that run on the physical OS threads. SQL Server

4   manually decides when to schedule these fibers (tasks) and on which OS thread

5   they should run. SQL also supports a "thread mode" in which a task is an actual

6   OS thread. In this mode, scheduling is largely done by the OS, although SQL does

7   use sync primitives to influence when code gets run.

8

9       There are a few places in this section where the behavior of a task relative

10   to a physical OS thread is specified. In those cases, the term "OS thread" is used

11   on purpose.

12

13       The threading abstraction and the synchronization abstraction are closely

14   related. That is, if a host 132 implements the threading abstraction they should

15   also implement the synchronization abstraction.

16

17   **1.6.1. Threads Not Under Host Control**

18       The threading abstraction allows the runtime 130 to create and manage

19   tasks through the host 132. However, the runtime 130 creates a few OS threads in

20   the process without going through the host 132ing API's. These threads are:

21   •    GC Threads. The number of these threads varies based on which collector

22         is running, the number of processors, etc.

23   •    A debug helper thread.

24   •    Internal wait pool threads. For example, the runtime 130 maintains a thread

25         for timers and a registered wait thread.

1   &bull;      Thread pool 'gate' thread.

2

3        In addition, unmanaged code entered from managed code may then create

4 OS threads and re-enter the runtime through managed code. Those OS threads are

5 also outside host control.

6

7        In this implementation, hosts do not take scheduling control of threads it

8 does not create or own. Hosts 132 should keep a map of threads with associated

9 tasks (hence under host control), and threads which are not under host control.

10 One exception may be the primary thread of execution started with the process.

11 This thread may not have a task associated, but it is under host control and can be

12 scheduled.

13

14        The following four interfaces comprise the threading abstraction of this

15 implementation:

16   &bull;      **IHostTaskManager**. Implemented by the host 132 and discovered by the

17          runtime 130 via IHostControl::GetHostManager. Allows the runtime 130

18          to work with tasks through the host 132 instead of using standard OS

19          threading or fiber API's. This interface provides methods for the runtime

20          130 to create and manage tasks, to notify the host 132 that the locale has

21          been changed through the BCL, to provide hooks for the host 132 to take

22          action when control is transferring in/out of the runtime 130 to run

23          unmanaged code and so on.

24          **IRuntimeTaskManager**. Hosts can use IRuntimeTaskManager to create

25          runtime tasks explicitly and to get the currently running runtime task.

1   •   **IHostTask.** The runtime 130 creates tasks using IHostTaskManager.

2        Every task created through IHostTaskManager has a representation on the

3        host 132 side and on the runtime 130 side. IHostTask is the task's

4        representation on the host 132 side while IRuntimeTask below for the

5        corresponding interface on the runtime 130 side. Once the task has been

6        created, IHostTask allows the runtime 130 to perform various actions on

7        the task including starting it, setting its priority, alerting it, and so on.

8   •   **IRuntimeTask.** The runtime's representation of the task. There is always a

9        corresponding instance of an IHostTask on the host 132 side. The host 132

10       calls IRuntimeTask to notify the runtime 130 about the state of the task,

11       including when the task is moving to or from the runnable state, or when it

12       is being aborted.

13

14       Fig. 4 shows an exemplary architectural relationship between runtime and

15 hosting application thread abstraction interfaces them:

16

17 **1.6.2. Creating Tasks**

18       As described above, each task running in a hosted environment has a

19 representation on the host 132 side (an instance of IHostTask) and a representation

20 on the runtime 130 side (an instance of IRuntimeTask). The host 132 task and

21 runtime task objects should be associated with each other in order for the host 132

22 and the runtime 130 to properly communicate about the underlying task. The two

23 task objects should be created and associated before any managed code can run on

24 the OS thread.

25

1       The process of creating a new task and its associated task objects can be

2    initiated either by the host 132 or by the runtime 130.

3

4    **<u>Runtime-initiated Task Creation</u>**

5       The runtime 130 will begin the process of creating new tasks when it is first

6    setting up a thread to run managed code. This occurs during the runtime's

7    initialization process, when a user explicitly creates a thread using the classes in

8    System.Threading or when the size of the thread pool grows.

9       In these scenarios, the steps taken to setup the new task include, for

10    example, the following:

11   •    The runtime 130 will create a runtime task and the corresponding

12        IRuntimeTask internally. Note: runtime tasks can also be "recycled" –

13        instead of creating a new runtime task from scratch, the runtime 130 may

14        choose to grab a task from its cache of recycled tasks. See

15        <u>IRuntimeTask::Reset</u> for details.

16   •    The runtime 130 will then call IHostTaskManager::CreateTask to create a

17        host 132 task to associate with the new runtime task.

18   •    The association is made when the runtime calls SetRuntimeTask on the

19        new IHostTask.

20

21       There is another scenario in which the runtime 130 will initiate task

22    creation. Threads that have not been initialized to run managed code may enter

23    the runtime 130 in scenarios such as COM interop and reverse PInvoke. In these

24    scenarios, the runtime 130 will ask the host 132 for the current host task as part of

25    initializing the thread. At this point the host 132 can refuse to return a task in

1   which case the runtime 130 will prevent thread from running managed code. The

2   steps, for example, are:

3   • A new thread wanders into the runtime.

4   • The runtime 130 calls IHostTaskManager::GetCurrentTask to get the current

5     host task.

6   • If a task is returned in step 2, the runtime 130 will create a runtime task object

7     (or use a recycled one) and an IRuntimeTask and call

8     IHostTask::SetRuntimeTask on the returned task.

9   • If no task is returned in step2, the runtime 130 will abort the thread.

10

11     **<u>Host-initiated Task Creation</u>**

12       Hosts may find it convenient at times to pre-create runtime tasks and their

13   associated IHostTasks before they are needed. This may be useful for hosts to

14   pre-initialize data structures for example.

15       Exemplary steps taken to setup a new task in this scenario are:

16   • The host 132 creates a host 132 task and the corresponding IHostTask

17     internally.

18   • The host 132 calls IRuntimeTaskManager::CreateTask to create a runtime task

19     to associate with the new host task.

20   • The runtime 130 calls IHostTaskManager:GetCurrentTask to get the currently

21     running host task.

22   • The runtime 130 then associates the runtime task with the host 132 task by

23     passing the new instance of IRuntimeTask to the SetRuntimeTask method on

24     the IHostTask returned in step 3.

25

1    **1.6.3. IHostTaskManager**

2

```
3    typedef enum {
4        WAIT_MSGPUMP = 0x1,
5        WAIT_ALERTABLE = 0x2,
6        WAIT_NOTINDEADLOCK = 0x4,
7    }WAIT_OPTION;
8
9    interface IHostTaskManager : IUnknown
10   {
11       HRESULT GetCurrentTask ([out] IHostTask **pTask);
12       HRESULT CreateTask ([in] DWORD stacksize,
13                           [in] LPTHREAD_START_ROUTINE pStartAddress,
14                           [in] PVOID pParameter,
15                   [out] IHostTask **ppTask);
16
17       HRESULT Sleep([in] DWORD dwMilliseconds,
18           [in] DWORD option);
19
20       HRESULT SwitchToTask([in] DWORD option);
21
22       HRESULT SetUILocale([in] LCID lcid);
23
24       HRESULT SetLocale([in] LCID lcid);
25
26       HRESULT GetUILocale([in] LCID lcid);
27
28       HRESULT GetLocale([in] LCID lcid);
29
30       HRESULT CallNeedsHostHook(
31           [in] SIZE_T target,
32           [out] BOOL * pbCallNeedsHostHook);
33
34       HRESULT LeaveRuntime([in] SIZE_T target);
35
36       HRESULT EnterRuntime();
37
38       HRESULT ReverseEnterRuntime();
39
40       HRESULT ReverseLeaveRuntime()
41
```

1    HRESULT BeginDelayAbort();

2

3    HRESULT EndDelayAbort();

4

5    HRESULT BeginThreadAffinity();

6

7    HRESULT EndThreadAffinity();

8

9    HRESULT SetRuntimeTaskManager([in] IRuntimeTaskManager

10   *pManager);

11   }

12   *1.6.3.1.      Enum: WAIT_OPTION*

13   • **WAIT_ALERTABLE**. Notifies the host 132 that the task should be awakened

14   if the runtime 130 alerts the task through IHostTask::Alert (see later in the

15   doc).   This flag is often passed to API's that accept timeout values (like

16   IHostTask::Join) in which case the host 132 is instructed to awake the task

17   before the timeout value if the runtime 130 alerts.

18   • **WAIT_MSGPUMP**. Notifies the host 132 that it should pump messages on

19   the current OS thread if the thread becomes blocked.   The runtime 130 will

20   only specify this value on an STA thread.

21   • **WAIT_NOTINDEADLOCK**.      Notifies   the   host   132   that   a   given

22   synchronization request won't be involved in a deadlock.   The runtime 130

23   passes this on "leaf" sync requests – i.e those requests that don't request

24   additional syncs.

25

26   *1.6.3.2.      GetCurrentTask*

27   Returns the host 132 task currently running on the OS thread from which

28   GetCurrentTask is called.

29

| Parameter | Description |
|---|---|
| pTask | [out] A pointer to the currently running task. If no task is currently running, NULL is returned.<br><br>If NULL is returned when the runtime 130 is attempting to initialize a new thread, the runtime 130 will stop initialization and will prevent that thread from running managed code. See "Runtime-initiated Task Creation" above. |

1

2    **HResults**

3    **See <u>Common HResults</u>**

4    **HOST_E_INVALIDOPERATION**. GetCurrentTask was called on an OS

5    thread the host 132 is not aware of.

6

7    *1.6.3.3.        Method:CreateTask*

8    Called by the runtime 130 to create a new host task. Newly created tasks

9    are in the suspended state.

10

| Parameter | Description |
|---|---|
| pStartAddress | [in] A pointer to the function to be executed by the task. |

| pParameter | [in] A pointer to the user data to be passed to the function. May be NULL. |
|---|---|
| ppTask | [out] A pointer to the host 132-created task or NULL if the task couldn't be created. The returned task is suspended until started with IHostTask::Start. |

1

2      **HResults**

3      **See Common HResults**

4      **E_OUTOFMEMORY**. Not enough resources were available to create the

5      task.

6

7      *1.6.3.4.     Sleep*

8      The runtime 130 calls this method to notify the host 132 that the current

9      task is going to sleep. This is called, for example, when Thread.Sleep is called

10     from user code.

11

| Parameter | Description |
|---|---|
| dwMilliseconds | [in] The number of milliseconds the thread will sleep. |
| Option | [in] Values from WAIT_OPTIONS indicating actions |

| | the host 132 should take if this operation blocks. See description for WAIT_OPTIONS above. |
|---|---|

1

2      **HResults**

3      **See Common HResults**

4

5      *1.6.3.5.      SwitchToTask*

6      SwitchToTask notifies the host 132 that it should switch out the current

7      task. The host 132 is free to switch in another task if desired.

8

| Parameter | Description |
|---|---|
| Option | [in]      Values      from WAIT_OPTIONS indicating actions the host 132 should take if this operation blocks.      See      description      for WAIT_OPTIONS      above.thread      or block its execution. |

9

10      **HResults**

11      **See Common HResults**

12

1     *1.6.3.6.       SetUILocale*

2     SetUILocale notifies the host 132 that the runtime 130 UI locale (UI

3     Culture in NLS+ terms) has been changed on the current task. This method is

4     called when the Thread.CurrentUICulture property is changed by user code. This

5     method exists to allow a host 132 to synchronize any mechanisms they have for

6     changing UI locale with a change made to the current UI locale done from

7     managed code.

8

9     If a host 132 does not allow the UI culture to be changed from managed

10     code, E_NOTIMPL should be returned from this method.

11

| Parameter | Description |
|---|---|
| Lcid | [in] The lcid that maps to the current task's new UI culture. |

12

13     **HResults**

14     **See <u>Common HResults</u>**

15     **E_NOTIMPL.** The host 132 does not allow the UI culture to be changed

16     from managed code.

17

18     *1.6.3.7.       SetLocale*

19     SetLocale notifies the host 132 that the runtime 130 locale (Culture in

20     NLS+ terms) has been changed on the current task. This method is called when

21     the Thread.CurrentCulture property is changed by user code. This method exists

1   to allow a host 132 to synchronize any mechanisms they have for changing locale

2   with a change made to the current locale done from managed code.

3

4        If a host 132 does not allow the culture to be changed from managed code,

5   E_NOTIMPL should be returned from this method.

6

| Parameter | Description |
|-----------|-------------|
| lcid | [in] The lcid that maps to the current task's new culture. |

7

8        **HResults**

9        **See Common HResults**

10       **E_NOTIMPL.** The host 132 does not allow the culture to be changed from

11  managed code.
12

13  *1.6.3.8.      CallNeedsHostHook*

14       CallNeedsHostHook and the (Return)Enter/(Return)Leave routines that

15  follow are the implementation of the requirement that host's be notified when calls

16  leave and enter the runtime 130 (see Entering and Leaving the Runtime).

17

18       For performance reasons, this implementation of the runtime 130 does an

19  analysis during JIT compilation of each PInvoke to determine if the call can be

20  inlined. CallNeedsHostHook provides a way for the host 132 to participate in this

21  decision making process. If the host 132 tells the runtime 130 that a given

22  PInvoke needs to be hooked, the runtime 130 will not inline the call. Common

1    reasons for requiring a hook are to adjust the floating point state or to receive

2    notification that a call is entering a state in which any locks taken or memory

3    requests made cannot be tracked by the host 132.

4

5       If a host 132 requests a call to be hooked the (Return)Enter/(Return)Leave

6    routines will be called on transitions into and out of the runtime 130.

7

| Parameter | Description |
|---|---|
| target | [in] The address within the mapped PE file of the unmanaged function that is about to be called. If the function about to be called is "well known" to the host 132 (i.e won't affect the execution environment with locks, etc...), the host 132 may elect to bypass the hook to improve performance. A common example of this is a call from a host-implemented managed library to an existing unmanaged routine that the host 132 either controls or fully understands. |
| pbCallNeedsHostHook | [out] A boolean value indicating whether the host 132 requires the call identified by target to |

| | be hooked. |
|---|---|

**HResults**

**See <u>Common HResults</u>**

### 1.6.3.9.       *LeaveRuntime*

Call sequences out of and back into the runtime 130 can be nested. Consider the following call sequence:

**Call1:** A managed Visual Basic program PInvokes to a method in a dll written in C.

**Call2:** The C program then calls a method in a C# DLL, thus re-entering managed code.

**Call3:** The C# method turns around and does a PInvoke of its own back out to unmanaged code.

**Call3:** Ends – we're back in the managed C# method.

**Call2:** Ends – we're back in the unmanaged C method.

**Call1:** Ends – we're back in the managed Visual Basic method.

In this case the outer level PInvoke reenters the runtime during the call (and goes back out again). The host 132ing api has two sets of enter/leave methods to differentiate nested calls from their "outer level" calls: Leave/EnterRuntime and ReverseLeave/ReverseEnterRuntime. The series of calls to LeaveRuntime, EnterRuntime, ReverseLeaveRuntime and ReverseEnterRuntime in these scenarios form a "stack" that lets the host 132 identify the nesting layers.

1       Fig. 5 shows exemplary managed/unmanaged code call sequences for

2    exiting/reverse-exiting and entering/reverse-entering managed code from/to

3    unmanaged code to/from unmanaged code.  As shown in Fig. 5:

4

5    • The first call that leaves the runtime 130 is an "outer level" call so the runtime

6        130 calls LeaveRuntime (assuming that the host 132 requested that the call be

7        hooked using CallNeedsHostHook).

8    • Because Call2 (from C to C#) is nested within the initial PInvoke, the runtime

9        130 calls ReverseEnterRuntime.

10   • Call3 leaves the runtime 130 again.  Because this is a new PInvoke

11       LeaveRuntime is called.

12   • Call3 now ends and EnterRuntime is called.

13   • Call2 now ends.  This is the ending of the call on which ReverseEnterRuntime

14       was called so the corresponding ReverseLeaveRuntime is called.

15   • The initial PInvoke now ends – EnterRuntime is called.
16

17       Fig. 7 shows an exemplary full stack of calls that were pushed and later

18   popped between runtime and hosting application thread abstraction interfacing

19   operations describe in the preceding example.  As shown, LeaveRuntime is called

20   when an "outer level" call is about to leave the runtime 130 and enter unmanaged

21   code.
22

| Parameter | Description |
|---|---|
| target | [in] The address within the mapped PE file of the unmanaged |

|  | function that is about to be called. Depending on the scenario the host 132 may use this address to take different actions depending on the function being called. |
|---|---|

1

2      **HResults**

3        See **Common HResults**

4            **E_OUTOFMEMORY**.    The host 132 attempted to allocate a

5      resource in the hook and failed.  The runtime 130 will fail the call.

6

7      *1.6.3.10.      EnterRuntime*

8            EnterRuntime is called when an outer level call is returning to the runtime

9      130.  EnterRuntime is called once for every corresponding LeaveRuntime.

10

11      **HResults**

12        See **Common HResults**

13            **E_OUTOFMEMORY**.    The host 132 attempted to allocate a

14      resource in the hook and failed.  The runtime 130 will fail the call.

15

16      *1.6.3.11.      ReverseLeaveRuntime*

17            ReverseLeaveRuntime is called when control leaves the runtime 130 for

18      unmanaged code while in the middle of an existing call (PInvoke).

19

20      **HResults**

1    See **Common HResults**

2         **E_OUTOFMEMORY**.    The host 132 attempted to allocate a

3    resource in the hook and failed.  The runtime 130 will fail the call.

4

5    *1.6.3.12.    ReverseEnterRuntime*

6         ReverseEnterRuntime is called when entering the runtime 130 from an

7    unmanaged call.    If    the    call    sequence    started    in    managed    code,

8    ReverseEnterRuntime will be called once for every ReverseLeaveRuntime.  Note

9    that calls can originate from unmanaged code and enter the runtime without being

10   nested.  In this case, no EnterRuntime, LeaveRuntime or ReverseLeaveRuntime

11   was seen and the ReverseLeave/ReverseEnter count will not balance.

12

13        **HResults**

14        See **Common HResults**

15        **E_OUTOFMEMORY**.    The host 132 attempted to allocate a

16   resource in the hook and failed.  The runtime 130 will fail the call.

17

18   *1.6.3.13.    BeginDelayAbort*

19        Notifies the host 132 of the beginning of a period of time in which the

20   current task cannot be aborted.  The host 132 should not abort the task until a

21   corresponding call to EndDelayAbort is received.  Calls to BeginDelayAbort

22   cannot be nested.

23

24        **HResults**

1    See **Common HResults**

2        **E_UNEXPECTED.**   A call to BeginDelayAbort was nested –

3    BeginDelayAbort had already been called and the corresponding

4    EndDelayAbort has been received yet.

5

6    *1.6.3.14.    EndDelayAbort*

7        Notifies the host 132 of the end of a period in which the current task cannot

8    be aborted.  A corresponding call to BeginDelayAbort should have been made on

9    this task for EndDelayAbort to have any affect.

10

11       **HResults**

12       See **Common HResults**

13       **E_UNEXPECTED**. EndDelayAbort was called without a corresponding

14   call to BeginDelayAbort.
15

16   *1.6.3.15.    BeginThreadAffinity*

17       Notifies the host 132 of the beginning of a period of time in which the

18   current task cannot be moved to a different OS Thread.  The task can be switched

19   out, but when it is switched in, it should be back on the same OS thread.  The task

20   should not be rescheduled to a different thread until a corresponding

21   EndThreadAffinity is received.  Calls to BeginThreadAffinity can be nested

22   because they apply to the current task.

23

24       **HResults**

25       See **Common HResults**

1

2 *1.6.3.16.      EndThreadAffinity*

3      Notifies the host 132 of the end of a period in which the current task cannot

4 be moved to a different OS thread.  A corresponding call to BeginThreadAffinity

5 should have been made on this task for EndThreadAffinity to have any affect.

6      **HResults**

7      See **Common HResults**

8      **E_UNEXPECTED**.      EndThreadAffinity    was    called    without    a

9 corresponding call to BeginThreadAffinity.
10

11 *1.6.3.17.      SetRuntimeTaskManager*

12      The  runtime  130  calls  this  method  to  set  a  pointer  to  the  runtime-

13 implemented task manager.

14

| Parameter | Description |
|---|---|
| pManager | [in] An interface pointer to the runtime-implemented task manager. This interface is of type IRuntimeTaskManager. |

15

16      **HResults**

17      See **Common HResults**

18

19 **1.6.4. IRuntimeTaskManager**
20

```
1    interface IRuntimeTaskManager : IUnknown
2    {
3        HRESULT CreateTask ([out] IRuntimeTask **pTask);
4        HRESULT GetCurrentTask ([out] IRuntimeTask **ppTask);
5        HRESULT SetUILocale([in] LCID lcid);
6        HRESULT SetLocale([in] LCID lcid);
7    }
8
```

9   *1.6.4.1.    CreateTask*

10      This method can be called by a host 132 to explicitly create a runtime task.

11  See <u>Creating Tasks</u> for scenarios in which CreateTask is used.

12

| Parameter | Description |
|-----------|-------------|
| ppTask | [out] A pointer to a new runtime task. NULL if the task couldn't be created. |

13

14      **HResults**

15      **See <u>Common HResults</u>**

16      **E_OUTOFMEMORY**.  Not enough resources were available to create the

17  task.

18

19  *1.6.4.2.    GetCurrentTask*

20      Returns the runtime task currently running on this OS thread.

21

| Parameter | Description |
|-----------|-------------|
| ppTask | [out] A pointer to the runtime task currently running on this OS |

|  | thread.  NULL is returned if no task is currently running. |
|---|---|

1

2      **HResults**

3      **See Common HResults**

4      *1.6.4.3.      SetUILocale*

5      SetUILocale notifies the runtime 130 that the host 132 has changed the

6  current UI locale on the current task.  This method exists to allow a host 132 to

7  synchronize any mechanisms they have for changing UI locale with those

8  provided through managed code (specifically the Thread.CurrentUICulture

9  property).  A call to SetUILocale changes the managed UI culture on the current

10  task.

11

| Parameter | Description |
|---|---|
| Lcid | [in] The lcid that maps to the current task's new UI culture. |

12

13      **HResults**

14      **See Common HResults**

15

16      *1.6.4.4.      SetLocale*

17      SetLocale notifies the runtime 130 that the host 132 has changed the current

18  locale on the current task.  This method exists to allow a host 132 to synchronize

19  any mechanisms they have for changing locale with those provided through

1      managed code (specifically the Thread.CurrentCulture property). A call to

2      SetLocale changes the managed culture on the current task.

3

| Parameter | Description |
|-----------|-------------|
| Lcid | [in] The lcid that maps to the current task's new culture. |

4

5      **HResults**

6      **See Common HResults**

7

8      **1.6.5. IHostTask**

9

10      interface IHostTask : IUnknown

11      {

12         HRESULT Start();

13         HRESULT Alert();

14         HRESULT Join([in] DWORD dwMilliseconds,

15             [in] DWORD option);

16         HRESULT SetPriority([in] int newPriority);

17         HRESULT GetPriority([out] int *pPriority);

18         HRESULT SetRuntimeTask([in] IRuntimeTask *pRuntimeTask);

19      }

1   *1.6.5.1.        Start*

2        Moves the task from the suspended state into a runnable state.  Start will

3   always return S_OK (unless a fatal error is encountered).

4

5        **HResults**

6        **See <u>Common HResults</u>**

7                                                                                      ·

8   *1.6.5.2.        Alert*

9        The runtime 130 calls Alert to wake the task so it can be aborted.  The

10  runtime 130 will call Alert when Thread.Abort is called from managed code, or as

11  a result of an AppDomain shutdown.  Calls to Alert are asynchronous – the host

12  132 should return immediately.  If the task isn't immediately alertable, it should

13  wake up the next time it enters an alertable state.  Note that this method only has

14  an affect on tasks where the runtime 130 has passed WAIT_ALERTABLE to

15  methods such as IHostTask::Join.

16                                   ·

17       **HResults**

18       **See <u>Common HResults</u>**

19

20  *1.6.5.3.        Join*

21       Blocks the calling task until this task terminates, the specified time interval

22  elapses or IHostTask::Alert is called (if the appropriate flag is passed).

23

| Parameter | Description |
|-----------|-------------|
| dwMilliseconds | [in] The number of milliseconds to |

| | |
|---|---|
| | wait until the task terminates. If this period elapses before the task terminates, the calling task will unblock. |
| Option | [in] A value from WAIT_OPTIONS. WAIT_OPTIONS::ALERTABLE directs the host 132 to wake the task if IHostTask::Alert is called before the timeout expires. |

1

2      **HResults**

3      See **Common HResults**

4

5    *1.6.5.4.     SetPriority*

6      The runtime 130 calls SetPriority when a user set's a thread's priority using

7   a managed API. Hosts are free to define the semantics of adjusting task priority.

8   For example, one implementation ignores this call as it does not allow user code to

9   adjust task priorities. SetPriority does not report whether the priority was adjusted

10  or not – use IHostTask::GetPriority to determine the tasks priority.

11

| Parameter | Description |
|---|---|
| newPriority | [in] The requested new priority for the task. The priority values are those defined by Win32's |

| | SetThreadPriority API. |
|---|---|

1

2 **HResults**

3 **See <u>Common HResults</u>**

4

5 *1.6.5.5.     GetPriority*

6 Returns the priority of the task.

7

| Parameter | Description |
|---|---|
| pPriority | [out] The task's priority.  The priority values are those defined by Win32's SetThreadPriority API. |

8

9 **HResults**

10 **See <u>Common HResults</u>**

11

12 *1.6.5.6.     SetRuntimeTask*

13 This method is called by the runtime 130 to associate a runtime task with a

14 host 132 task created using IHostTaskManager::CreateTask.  See <u>Creating Tasks</u>

15 for scenarios in which SetRuntimeTask is called.

16

| Parameter | Description |
|---|---|
| pRuntimeTask | [in] An interface pointer to the runtime task to associate with this host |

| | task. |
|---|---|

1

2      See **Common HResults**

3

4    **1.6.6. IRuntimeTask**

5

6    interface IRuntimeTask: IUnknown

7    {

8        HRESULT SwitchIn([in] HANDLE threadHandle);

9        HRESULT SwitchOut();

10        HRESULT GetMemStats([out] COR_GC_THREAD_STATS

11    *pMemUsage);

12      HRESULT Reset([in] BOOL fFull);

13        HRESULT ExitTask();

14        HRESULT Abort();

15        HRESULT RudeAbort();

16        HRESULT NeedsPriorityScheduling([out] BOOL *

17    pbNeedsPriorityScheduling);

18        HRESULT YieldTask();

19        HRESULT LocksHeld([out] SIZE_T *pLockCount);

20        HRESULT SetTaskIdentifier([in] DWORD asked);

21    }

22

23      Fig. 6 is a block diagram showing exemplary task scheduling of runtime

24  tasks that are treated as fibers scheduled on OS threads by a host application 132.

25  At any point in time, a runtime task is always in one of two primary states:

26  running or waiting to be run.  For example:

27  •      Runtime tasks are in the unscheduled, or "not-running" state when they are

28        first created.

29  •      Hosts call IRuntimeTask::SwitchIn to transition a task onto an OS thread.

30        At this point, the task runs managed code.

1   •   At some later point in time, the host 132 may decide to "switch the task

2       out", or remove it from the thread and place it back in a non-running state.

3       Hosts may choose to do this if the running task blocks on synchronization

4       primitives, waits for I/O to complete, and so on. Runtime tasks are moved

5       to the unscheduled state when a host 132 calls IRuntimeHost::SwitchOut.

6   •   Later, a host 132 may elect to schedule the task again by calling SwitchIn

7       again. In the general case, the host 132 may schedule the task on any OS

8       thread.

9   •   However, there are scenarios in which the runtime 130 requires thread

10      affinity for a given task. Periods of thread affinity are defined when the

11      runtime 130 calls IHostTaskManager::BeginThreadAffinity and the

12      corresponding EndThreadAffinity. In this particular diagram, the host 132

13      should schedule the task back on OS Thread 1 if the task requires thread

14      affinity when time comes to reschedule it.

15   •   Eventually, a runtime task will end permanently. This generally happens

16      when the code running on the task reaches it natural end. In this scenario,

17      the host 132 calls IRuntimeTask::ExitTask to destroy the clr task. Runtime

18      tasks can also be "recycled". Instead of calling ExitTask, a host 132 can

19      call IRuntimeTask::Reset to restate a runtime task to a clean state. The

20      runtime 130 will then reuse this task in the future instead of creating a new

21      one. See IRuntimeTask::Reset for details.
22

23   *1.6.6.1.*     *SwitchIn*

24      This method notifies the runtime 130 that the task is now in the runnable

25  state. A handle to the OS thread that the task has been scheduled on is passed as a

1   parameter. If impersonation has been done on this thread, it should be reverted

2   (RevertToSelf) before switching in the runtime task.

3        SwitchIn cannot be called twice without a corresponding SwitchOut.

4

| Parameter | Description |
|---|---|
| threadHandle | [in] A handle to the physical thread this task is running on. |

5

6        **HResults**

7        **See Common HResults**

8

9   *1.6.6.2.       SwitchOut*

10       SwitchOut notifies the runtime 130 that the task has been removed from the

11  runnable state.

12

13       **See Common HResults**

14  *1.6.6.3.       GetMemStats*

15       Returns memory statistics for the task.

16

| Parameter | Description |
|---|---|
| pMemUsage | [out]. A structure containing information about the memory usage of this task, including the number of bytes allocated. See gchost.idl for details. |

1

2      **HResults**

3      **See <u>Common HResults</u>**

4

5   *1.6.6.4.     Reset*

6      The runtime 130 can reuse previously created tasks in order to avoid the

7  overhead of repeatedly creating new tasks each time a fresh task is needed. Hosts

8  enable this task reuse feature by calling Reset instead of ExitTask when finished

9  with a runtime task.

10

11      For example, one of the scenarios described above in <u>Creating Tasks</u> works

12  roughly as follows:

13    (1) A new thread wanders into the runtime 130

14    (2) The runtime 130 creates a new runtime task object and calls

15         IHostTask::SetRuntimeTask on the current host task (obtained through

16         IHostTaskManager::GetCurrentTask).

17    (3) The task executes normally.

18    (4) At some point, the task reaches its end so the host 132 destroys it with

19         IRuntimeTask::ExitTask.

20

21      Using Reset can change the above scenario as follows: Instead of

22  destroying the task in Step #4, the host 132 can call Reset to reset the runtime task

23  to a clean state. The host 132 will then decouple the task from the associated host

24  task (which the host 132 may choose to cache as well). Then in Step #2, instead

1   of creating a new runtime task, the runtime 130 will grab a recycled task and call

2   IHostTask::SetRuntimeTask on the current host task.

3

4       This approach works very well in scenarios where the host 132 has a pool

5   of worker tasks they are reusing themselves.  For purposes of discussion, a worker

6   task, as is a task, refers to any executing portion of program module code in

7   system memory 106 or logically coupled to system memory 106.  When a host

8   132 is done with one of its worker task, it calls IRuntimeTask::ExitTask to destroy

9   the runtime task as well.

10

| Parameter | Description |
|---|---|
| fFull | [in] A Boolean value indicating whether a full reset should be done.  If false, the runtime 130 will reset all security and locale information on the task.  If true, the runtime 130 will reset all thread-relative statics and all slots reserved through calls to Thread.Alloc(Named)DataSlot in addition to the security and locale information. |

11

12      **HResults**

13      **See <u>Common HResults</u>**

14

1    *1.6.6.5.        ExitTask*

2        Called by the host 132 to notify the runtime 130 that the task is ending.

3    ExitTask attempts a clean shutdown of the task – it is This is the equivalent to

4    getting a Win32 thread detach on a dll.

5

6        **HResults**

7        **See Common HResults**

8

9    *1.6.6.6.        Abort*

10       Called by the host 132 to abort the task.   The runtime 130 raises a

11   System.Threading.ThreadAbortException when this method is called.   Abort

12   returns after the exception processing is initiated - it does not wait for user code

13   such as finalizers and exception code to finish.  As such, calls to Abort return

14   quickly.

15

16       **HResults**

17       **See Common HResults**

18   *1.6.6.7.      RudeAbort*

19       Called by the host 132 to unconditionally abort a task.  The runtime 130

20   aborts the task immediately.   Finalizers and exception handling code are not

21   guaranteed to be run.

22

23       **HResults**

24       **See Common HResults**

25

1    *1.6.6.8.        NeedsPriorityRescheduling*

2        Hosts 132 call NeedsPriorityRescheduling when a task is being switched

3    out.  The return value from this method provides a hint to the host 132 as the

4    priority with which to reschedule the task.  The runtime 130 will return true (high

5    priority reschedule) in situations where the task preparing for garbage collection.

6    By quickly rescheduling the task, the host 132 minimizes the possibility that

7    garbage collection will be delayed to the point where memory usage becomes an

8    issue.

9

| Parameter | Description |
|---|---|
| pbNeedsPriorityScheduling | [out] A value of true indicates that the host 132 should make every attempt to reschedule this task as soon as possible.  False indicates no special scheduling treatment for the task. |

10

11        **HResults**

12        **See Common HResults**

13    *1.6.6.9.      YieldTask*

14        The runtime 130 will attempt to get the task to a state where it will yield.

15    This method is intended to cause long running code to give up the CPU to other

16    tasks.  The runtime 130 does not guarantee that the task will eventually yield.

17

18        **HResults**

1    **See <u>Common HResults</u>**

2

3    *1.6.6.10.    LocksHeld*

4    Returns the number of locks that are currently held on the task.

5

| Parameter | Description |
|-----------|-------------|
| pLockCount | [out] The number of locks currently being held on the task. |

6

7    **HResults**

8    **See <u>Common HResults</u>**

9

10    *1.6.6.11.    SetTaskIdentifier*

11    SetTaskIdentifier is used by the host 132 to associate an opaque identifier

12    with the task to achieve better host-clr integration in the debugger. This id allows

13    the debugger to identify that a runtime 130 call stack and a host 132 call stack are

14    associated and therefore can be merged to present a unified view to the user of the

15    debugger.

16

| Parameter | Description |
|-----------|-------------|
| Asked | [in] The id to associate with the task. The value of this id has no semantics in the runtime 130 – it is simply passed through to debuggers. |

1

2        **HResults**

3        **See <u>Common HResults</u>**

4

5    **1.6.7. IHostGCManager**

6

7    interface IHostGCManager : IUnknown
8    {
9        HRESULT ThreadIsBlockingForSuspension();
10       HRESULT SuspensionStarting();
11       HRESULT SuspensionEnding(DWORD Generation);
12   }
13

14   *1.6.7.1.        Method ThreadIsBlockingForSuspension*

15       The runtime calls this method to notify the host 132 that the thread making

16   the call is about to block, perhaps for a GC or other suspension.  This gives the

17   host 132 an opportunity to re-schedule the thread for unmanaged tasks.

18

19   *1.6.7.2.        Method SuspensionStarting*

20       The runtime calls this method to notify the host 132 it is beginning a thread

21   suspension for a GC or other suspension.

22       In this implementation, this thread is not rescheduled.

23   *1.6.7.3.        Method SuspensionEnding*

24       The runtime 130 calls this method to notify the host 132 that it is resuming

25   threads after a GC or other suspension.      In this implementation, this thread is

26   not rescheduled.

27

| Parameter | Description |
|-----------|-------------|

| generation | [in] The garbage collection generation just finishing, from which the thread is resuming. |
|---|---|

1

### 1.6.8. IHostPolicyManager

3

```
interface IHostPolicyManager: IUnknown
{
    HRESULT OnDefaultAction(
        [in] EClrOperation operation,
        [in] EPolicyAction action);

    HRESULT OnTimeout(
        [in] EClrOperation operation,
        [in] EPolicyAction action);

    HRESULT OnFailure(
        [in] EClrFailure failure,
        [in] EPolicyAction action);
}
```

18

### *1.6.8.1. OnDefaultAction*

| Parameter | Description |
|---|---|
| Operation | [in] The operation the runtime 130 is performing for this action |
| Action | [in] The action the runtime 130 is performing |

20

21 Called by the runtime 130 on the host 132 when the default action, based

22 on IRuntimePolicyManager::SetDefaultAction, is about to be taken in response to

23 thread abort, appdomain unload, etc.

1

2 *1.6.8.2.    OnTimeout*

| Parameter | Description |
|---|---|
| Operation | [in] The operation the runtime 130 is performing for this action |
| Action | [in] The action the runtime 130 is performing |

3

4    Called by the runtime 130 on the host 132 when an escalation action, based

5 on IRuntimePolicyManager::SetActionOnTimeout, is about to be taken in

6 response to thread abort, appdomain unload, etc.

7

8 *1.6.8.3.    OnFailure*

| Parameter | Description |
|---|---|
| Failure | [in] The type of resource allocation or reclamation failure which occurred. |
| Action | [in] The action the runtime 130 is performing |

9

10    Called by the runtime 130 on the host 132 when a resource allocation or

11 reclamation          failure          occurs,          based          on

12 IRuntimePolicyManager::SetActionOnFailure.

13

1    1.7. ThreadPool Abstraction Interfaces

2        The thread pool abstraction consists of a single interface

3    (IHostThreadPoolManager) implemented by the host 132.  The runtime 130 uses

4    this interface to configure the thread pool and queue work items to it.

5

6    **1.7.1. IHostThreadpoolManager**
7
8    interface IHostThreadpoolManager : IUnknown
9        {
10              HRESULT QueueUserWorkItem(
11                  [in] LPTHREAD_START_ROUTINE Function,
12                  [in] PVOID Context,
13                  [in] ULONG Flags);
14              HRESULT SetMaxThreads(
15                  [in] DWORD MaxThreads);
16              HRESULT GetMaxThreads(
17                  [out] DWORD *pdwMaxThreads);
18              HRESULT GetAvailableThreads(
19                  [out] DWORD *pdwAvailableWorkerThreads);
20              HRESULT SetMinThreads(
21                  [in] DWORD MinThreads);
22              HRESULT GetMinThreads(
23                  [out] DWORD *MinThreads);
24          }
25

26    *1.7.1.1.      QueueUserWorkItem*

27        QueueUserWorkItem wraps the Win32 function of the same name.  All

28    parameters are identical.

29

30        **HResults**

31        **See Common HResults**

32

1   *1.7.1.2.        SetMaxThreads*

2        SetMaxThreads allows the runtime 130 to set the maximum number of

3   threads in the thread pool.  Hosts 132 are not required to allow the size of the pool

4   to be configured in this way – it's likely that some hosts will want exclusive

5   control over the thread pool size for internal reasons including implementation,

6   performance and scalability.    In such a scenario, a host should return

7   E_NOTIMPL.

8

| Parameter | Description |
|-----------|-------------|
| MaxThreads | [in] The maximum number of threads to create in the pool. |

9

10       **HResults**

11       **See Common HResults**

12       **E_NOTIMPL.**  The host 132 doesn't provide an implementation of this

13   method.

14

15  *1.7.1.3.        GetMaxThreads*

16       This method returns the maximum number of threads that will be created in

17   the thread pool.

| Parameter | Description |
|-----------|-------------|
| MaxThreads | [out] The maximum number that will be created in the pool. |

18

19       **HResults**

1    See **Common HResults**

2    **E_NOTIMPL.**  The host 132 doesn't provide an implementation of this

3    method.

4

5    *1.7.1.4.        GetAvailableThreads*

6        This method returns the number of threads in the pool that are not currently

7    servicing requests.

8

| Parameter | Description |
|---|---|
| AvailableThreads | [out] The number of threads that are currently available to service user requests. |

9

10    **HResults**

11    See **Common HResults**

12    **E_NOTIMPL.**  The host 132 doesn't provide an implementation of this

13    method.

14    *1.7.1.5.      SetMinThreads*

15        This method allows the runtime to establish the minimum number of

16    threads created in the pool.

17

| Parameter | Description |
|---|---|
| MinThreads | [in] The minimum number of threads to create in the pool |

1

*1.7.1.6.*      *GetMinThreads*

This method allows the runtime to query the host 132 for the minimum number of threads available in the pool.

| Parameter | Description |
|---|---|
| MinThreads | [out] The minimum number of threads the host 132 provides in the pool. |

1.8. I/O Completion Abstraction Interfaces

The I/O Completion abstraction consists of two interfaces: A host-implemented manager (IHostIOCompletionManager) and a corresponding runtime 130-implemented manager (IRuntimeIOCompletionManager). The runtime 130 calls IHostIOCompletionManager to bind handles to a host-provided completion ports and the host 132 calls IRuntimeIOCompletionManager to notify the runtime 130 that a given I/O request has completed.

**1.8.1. IHostIoCompletionManager**


interface IHostIoCompletionManager : IUnknown

{

```
    HRESULT CreateIoCompletionPort(
        [out] HANDLE *phPort);
    HRESULT CloseIoCompletionPort(
        [in] HANDLE hPort);
```

```
1    HRESULT SetMaxThreads(
2        [in] DWORD dwMaxIOCompletionThreads);
3    HRESULT GetMaxThreads(
4        [out] DWORD *pdwMaxIOCompletionThreads);
5    HRESULT GetAvailableThreads(
6        [out] DWORD *pdwAvailableIOCompletionThreads);
7    HRESULT GetHostOverlappedSize(
8        [out] DWORD * pcbSize);
9    HRESULT SetRuntimeIoCompletionManager(
10       [in] IRuntimeIoCompletionManager * pManager);
11   HRESULT InitializeHostOverlapped(
12       [in] void * pvOverlapped);
13   HRESULT Bind(
14       [in] HANDLE hPort,
15       [in] HANDLE hHandle);
16   HRESULT SetMinThreads(
17       [in] DWORD dwMinIOCompletionThreads);
18   HRESULT GetMinThreads(
19       [out] DWORD *dwMinIOCompletionThreads);
20

21   }
```

22

### 1.8.1.1.    *CreateIoCompletionPort*

24    Creates a new IO Completion port through the host 132. I/O operations are

25 bound to the new port using IHostIoCompletionManager::Bind. Status is reported

26 back to the runtime 130 by calls to IRuntimeIoCompletionManager::OnComplete.

27

| Parameter | Description |
|---|---|
| phPort | [out] The newly created I/O Completion port, or 0 if no port could be created. |

28

29    **HResults**

1    **See Common HResults**

2    **E_OUTOFMEMORY**. Not enough resources were available to create the

3    task.

4

5    *1.8.1.2.        CloseIoCompletionPort*

6    Closes a port created with CreateIoCompletionPort.

7

| Parameter | Description |
|---|---|
| hPort | [out] The I/O completion port to close.   The port should have been created with a previous call to CreateIoCompletionPort. |

8

9    **HResults**

10   **See Common HResults**

11   **E_INVALIDARG**. An invalid port was passed.

12

13   *1.8.1.3.        SetMaxThreads*

14   SetMaxThreads allows the runtime 130 to set the maximum number of

15   threads that will be made available to service requests on I/O Completion ports.

16   Hosts are not required to allow the number of available threads to be configured in

17   this way – it's likely that some hosts will want exclusive control over the number

18   of threads size for internal reasons including implementation, performance and

19   scalability.  Hosts should return E_NOTIMPL in this case.

1

| Parameter | Description |
|---|---|
| dwMaxIOCompletionThreads | [in] The maximum number of threads to create in order to handle I/O completion requests. |

2

3        **HResults**

4     **See Common HResults**

5        **E_NOTIMPL.** The host 132 doesn't provide an implementation of this

6    method.

7

8    *1.8.1.4.      GetMaxThreads*

9        This method returns the maximum number of threads that will be created to

10   handle I/O completion requests.

11

| Parameter | Description |
|---|---|
| pdwMaxIOCompletionThreads | [out] The maximum number of threads that will be created in the pool to handle I/O completion requests. |

12

13       **HResults**

14    **See Common HResults**

1    **E_NOTIMPL.**   The host 132 doesn't provide an implementation of this

2    method.

3

4    *1.8.1.5.        GetAvailableThreads*

5       This method returns the number of I/O completion threads that are not

6    currently servicing requests.

7

| Parameter | Description |
|---|---|
| pdwAvailableIOCompletionThreads | [out] The number of I/O completion threads currently available to service user requests. |

8

9    **HResults**

10   **See <u>Common HResults</u>**

11   **E_NOTIMPL.**   The host 132 doesn't provide an implementation of this

12   method.

13

14   *1.8.1.6.        GetHostOverlappedSize*

15      All asynchronous I/O calls made to Win32 contain an OVERLAPPED

16   structure that provides data like file pointer position.  Often times, applications

17   making async I/O calls append custom information to the end of the

18   OVERLAPPED structure to maintain custom state about the request.

19

1           IHostIOCompletionManager allows the host 132 to append this custom

2     data with calls to GetHostOverlappedSize and InitializeHostOverlapped.

3     GetHostOverlappedSize is called by the runtime 130 to determine the size of any

4     custom data a host 132 wants to attach to the OVERLAPPED structure. This

5     method is only called once - the size of the host 132's custom data should be the

6     same for every request.

7

| Parameter | Description |
|---|---|
| pcbSize | [out] The number of bytes the runtime 130 should allocate at the end of the OVERLAPPED structure for the host 132's custom data. DO NOT include the size of the OVERLAPPED structure itself in this value. |

8

9     **HResults**

10     **See <u>Common HResults</u>**

11

12     *1.8.1.7.*        *InitializeHostOverlapped*

13          The runtime 130 calls InitializeHostOverlapped before every async I/O

14     request to give the host 132 a chance to initialize any custom data appended to the

15     OVERLAPPED structure.

16

| Parameter | Description |
|---|---|

| pvOverlapped | [in] A pointer to the beginning of the OVERLAPPED structure. Hosts should offset by sizeof(OVERLAPPED) to get to the start of their custom data block. |
|---|---|

1

2      **HResults**

3      See **Common HResults**

4          **E_OUTOFMEMORY**. An out of memory error occurred while the

5      host 132 was trying to allocate memory as part of initializing the data

6      structure. The runtime 130 will return an error to the user and fail the call.

7

8      *1.8.1.8.        SetRuntimeIOCompletionManager*

9          This method provides the host 132 with an interface pointer to the runtime

10     130's implementation of IRuntimeIOCompletionManager. The host 132 calls this

11     interface to notify the runtime 130 when an I/O request has completed.

12

| Parameter | Description |
|---|---|
| pManager | [in] A pointer to the runtime 130's implementation of IRuntimeIOCompletionManager. |

13

14     **HResults**

15     See **Common HResults**

1

2      *1.8.1.9.      Bind*

3      This method binds a handle to a completion port previous created with

4   IHostIoCompletionManager::CreateIOCompletionPort.    When the I/O request

5   completes,        the        host        132        should        call

6   IRuntimeIOCompletionManager::OnComplete.

7

| Parameter | Description |
|-----------|------------|
| hPort | [in] The port to bind to.  If NULL is passed, the default IO completion port is bound.  Ports are created by                                                              calling IHostIoCompletionManager::CreateIOCompletionPort. |
| hHandle | [in] The OS handle to bind to the completion port. |

8

9      **HResults**

10      **See Common HResults**

11

12      *1.8.1.10.      SetMinThreads*

13      This method allows the runtime to establish the minimum number of IO

14   completion threads created by the host 132.

15

| Parameter | Description |
|-----------|------------|
| MinThreads | [in]  The  minimum  number  of |

| | threads to create in for IO completion requests. |
|---|---|
| | |

1

2 *1.8.1.11.*     *GetMinThreads*

3      This method allows the runtime to query the host 132 for the minimum

4 number of threads available for IO completion requests.

5

| Parameter | Description |
|---|---|
| MinThreads | [out] The minimum number of threads the host 132 provides to service IO completions. |

6

7 **1.8.2. IRuntimeIoCompletionManager**
8
9     interface IRuntimeIoCompletionManager : IUnknown
10     {
11       HRESULT OnComplete(
12         [in] DWORD dwErrorCode,
13         [in] DWORD NumberOfBytesTransferred,
14         [in] void * pvOverlapped);
15     }
16

17 *1.8.2.1.*     *OnComplete*

18      OnComplete is the completion callback for I/O requests started with

19 IHostIOCompletionManager::Bind. Hosts pass an HRESULT to OnComplete that

20 describes the outcome of the bind operation. The runtime 130 is equipped to

21 handle requests that have been terminated before completing successfully.

22 Resource pressure in the host 132 may sometimes cause the host 132 to stop a

1   thread that has a pending I/O request. In this scenario, the host 132 passes an

2   HRESULT (HOST_E_INTERRUPTED) to OnComplete that indicates the request

3   was terminated prematurely.

4

| Parameter | Description |
|---|---|
| dwErrorCode | [in] An HRESULT indicating why OnComplete was called.<br><br>Calls that have completed successfully return S_OK.<br><br>Calls that have failed for an unrecoverable, catastrophic reason return E_FAIL.<br><br>Calls that have been terminated before completing successfully return HOST_E_INTERRUPTED. |
| NumberOfBytesTransferred | [in] The number of bytes that have been transferred during this I/O request. |
| pvOverlapped | [in] A pointer to the OVERLAPPED structure associated with this request when IHostIOCompletionManager::Bind was called. |

5

1    **HResults**

2    **See <u>Common HResults</u>**

3

4    1.9. Synchronization Abstraction Interfaces

5    The synchronization abstraction consists of a set of interfaces that allow the

6    runtime 130 to create various synchronization primitives through the host 132 and

7    "manager" interfaces on both the runtime 130 and host side:

8    • **IHostSyncManager**. Implemented by the host 132 and discovered by the

9    runtime 130 via IHostControl::GetHostManager.  Allows the runtime 130

10    to create sync primitives through the host 132 instead of using standard OS

11    API's.

12    • **IHostCriticalSection**. A host-implemented critical section.

13    • **IHostAutoEvent**. A host-implemented auto-reset event.

14    • **IHostManualEvent**. A host-implemented manual-reset event.

15    • **IHostSemaphore**.  A host-implemented semaphore.

16    • **IRuntimeSyncManager**. A runtime 130-implemented interface that allows

17    the host 132 to implement deadlock detection on lazily created sync objects

18    created by the runtime 130.

19    All of these interfaces, except IRuntimeSyncManager are implemented by

20    the host 132.

21

22    **1.9.1. IHostSyncManager**
23
24    interface IHostSyncManager : IUnknown
25    {
26    HRESULT SetRuntimeSyncManager([in] IRuntimeSyncManager
27    *pManager);

```
1    HRESULT CreateCrst([out] IHostCriticalSection** ppCrst);
2    HRESULT CreateCrstWithSpinCount ([in] DWORD dwSpinCount,
3                        [out] IHostCriticalSection** ppCrst);
4
5    HRESULT CreateAutoEvent([out] IHostAutoEvent **ppEvent);
6    HRESULT CreateManualEvent([in] BOOL bInitialState,
7                [out] IHostManualEvent **ppEvent);
8
9    HRESULT CreateMonitorEvent([in] SIZE_T Cookie,
10                   [out] IHostAutoEvent **ppEvent);
11
12   HRESULT CreateRWLockWriterEvent([in] SIZE_T Cookie,
13                   [out] IHostAutoEvent **ppEvent);
14
15   HRESULT CreateRWLockReaderEvent([in] BOOL bInitialState,
16                [in] SIZE_T Cookie,
17                [out] IHostManualEvent **ppEvent);
18
19       HRESULT CreateSemaphore([in] DWORD dwInitial,
20               [in] DWORD dwMax,
21               [out] IHostSemaphore ** ppSemaphore);
22   }
23
```

24   *1.9.1.1.        SetRuntimeSyncManager*

25   Called by the runtime 130 to sets the runtime 130-implemented manager

26   that corresponds to this host manager.

27

| Parameter | Description |
|-----------|-------------|
| pManager | [in] A pointer to the IRuntimeSyncManager provided by the runtime 130. |

28

29   **HResults**

30   **See <u>Common HResults</u>**

1

### 1.9.1.2. *CreateCrst*

Creates a critical section, for example, such as the WIN32 InitializeCriticalSection interface.

| Parameter | Description |
|-----------|-------------|
| ppCrst | [out] A pointer to the host 132 implemented critical section or NULL if a critical section couldn't be created. |

**HResults**

**See Common HResults**

**E_OUTOFMEMORY**. Not enough resources were available to create the critical section.

### 1.9.1.3. *CreateCrstWithSpinCount*

Creates a critical section with a spin count, for example, such as the WIN32 InitializeCriticalSectionAndSpinCount.

| Parameter | Description |
|-----------|-------------|
| dwSpinCount | [in] Specifies the spin count for the critical section object. The semantics of this parameter are the same as the same-named parameter in Win32's |

|  | InitializeCriticalSectionAndSpinCount. |
|---|---|
| ppCrst | [out] A pointer to the host 132 implemented critical section or NULL if a critical section couldn't be created. |

1

2      **HResults**

3      **See Common HResults**

4      **E_OUTOFMEMORY.** Not enough resources were available to create the

5 critical section.

6

7 *1.9.1.4.          CreateAutoEvent*

8      Creates an auto-reset event.  This method mirrors the Win32 function

9 CreateEvent with bManualReset set to false.

10

| Parameter | Description |
|---|---|
| ppEvent | [out] A pointer to the host 132 implemented auto reset event or NULL if an event couldn't be created. |

11

12      **HResults**

13      **See Common HResults**

14      **E_OUTOFMEMORY.** Not enough resources were available to create the

15 event.

16

1    *1.9.1.5.       CreateManualEvent*

2        Creates a manual-reset event.

3

| Parameter | Description |
|---|---|
| bInitialState | [in] The semantics of this parameter are the same as the same-named parameter in Win32's CreateEvent. True means the object is signaled, false means it is not signaled. |
| ppEvent | [out] A pointer to the host 132 implemented manual reset event or NULL if an event couldn't be created. |

4

5        **HResults**

6        **See <u>Common HResults</u>**

7        **E_OUTOFMEMORY**.  Not enough resources were available to create the

8    event.

9

10   *1.9.1.6.       CreateMonitorEvent*

11       Creates an auto-reset event that the runtime 130 uses to implement the Base

12   Class Library (BCL) Monitor class.

13

| Parameter | Description |
|---|---|
| cookie | [in] A cookie that the host 132 would like to associate with the monitor. |

|  |  |
|---|---|
|  | The host 132 can use this cookie to determine which task is waiting on the monitor using IRuntimeSyncManager::GetMonitorOwner. |
| ppEvent | [out] A pointer to the host 132 implemented auto reset event or NULL if an event couldn't be created. |

1

2      **HResults**

3      **See Common HResults**

4      **E_OUTOFMEMORY**.  Not enough resources were available to create the

5  event.

6

7      *1.9.1.7.*      *CreateRWLockWriterEvent*

8      Creates an auto-reset event that the runtime 130 uses to implement a writer

9  lock in a Reader/Writer lock.

10

| Parameter | Description |
|---|---|
| cookie | [in] A cookie that the host 132 would like to associate with the writer lock.  The host 132 can use this cookie to determine which tasks are waiting on the lock using the ReaderWriterLock iteration methods on |

| | IRuntimeSyncManager. |
|---|---|
| ppEvent | [out] A pointer to the host 132 implemented auto reset event or NULL if an event couldn't be created. |

1

2     **HResults**

3     See **Common HResults**

4     **E_OUTOFMEMORY**. Not enough resources were available to create the

5 event.

6

7     *1.9.1.8.*     *CreateRWLockReaderEvent*

8     Creates a manual-reset event that the runtime 130 uses to implement a

9 reader lock in a Reader/Writer lock.

10

| Parameter | Description |
|---|---|
| bInitialState | [in] The semantics of this parameter are the same as the same-named parameter in Win32's CreateEvent. True means the object is signaled, false means it is not signaled. |
| cookie | [in] A cookie that the host 132 would like to associate with the reader lock. The host 132 can use this cookie to determine which tasks are waiting on |

| | |
|---|---|
| | the lock using the ReaderWriterLock iterations methods on IRuntimeSyncManager. |
| ppEvent | [out] A pointer to the host 132 implemented manual reset event or NULL if an event couldn't be created. |

1

2     **HResults**

3     **See <u>Common HResults</u>**

4     **E_OUTOFMEMORY**. Not enough resources were available to create the

5   event.

6

7   *1.9.1.9.*      *CreateSemaphore*

8     Creates a host-implemented semaphore. This method mirrors Win32's

9   CreateSemaphore.

10

| Parameter | Description |
|---|---|
| dwInitial | [in] Specifies the initial count for the semaphore. This parameter has the same semantics as the lInitialCount parameter to Win32's CreateSemaphore. |
| dwMax | [in] Specifies the maximum count for the semaphore. This |

| | |
|---|---|
| | parameter has the same semantics as the lMaximumCount parameter to Win32's CreateSemaphore. |
| ppSemaphore | [out] A pointer to the host 132 implemented semaphore or NULL if a semaphore couldn't be created. |

1

2    **HResults**

3    **See <u>Common HResults</u>**

4    **E_OUTOFMEMORY**. Not enough resources were available to create the

5    semaphore.

6

7    **1.9.2. IHostCrst**

8

```
9    interface IHostCrst : IUnknown
10   {
11       HRESULT Enter([in] DWORD option);
12       HRESULT Leave();
13       HRESULT TryEnter([in] DWORD option,
14               [out] BOOL *pbSucceeded);
15       HRESULT SetSpinCount([in] DWORD dwSpinCount);
16   }
```

17

18    *1.9.2.1.    Enter*

19    Attempts to enter the critical section. Enter will not return until the critical

20    section is entered (e.g., see Win32 EnterCriticalSection)

21

| Parameter | Description |
|---|---|
| Option | [in]        Values        from |

| | |
|---|---|
| | WAIT_OPTIONS indicating actions the host 132 should take if this operation blocks. See description for WAIT_OPTIONS above. |

1

2      **HResults**

3      **See Common HResults**

4

5   *1.9.2.2.      Leave*

6      Leaves the critical section (e.g., see Win32 LeaveCriticalSection).

7

8      **HResults**

9      **See Common HResults**

10

11  *1.9.2.3.      TryEnter*

12      Attempts to enter the critical section. TryEnter returns immediately and

13  indicates whether the critical section was entered or not (e.g., see Win32

14  TryEnterCriticalSection).

15

| Parameter | Description |
|---|---|
| Option | [in]      Values      from WAIT_OPTIONS indicating actions the host 132 should take if this operation blocks. See description for |

| | WAIT_OPTIONS above. |
|---|---|
| pbSucceeded | [out] Indicates whether the critical section could be entered. True if entered, false if not. |

1

2      **HResults**

3      **See Common HResults**

4

5      *1.9.2.4.      SetSpinCount*

6      Sets a spin count for the critical section.

| **Parameter** | **Description** |
|---|---|
| dwSpinCount | [in] Specifies the spin count for the critical section object. The semantics of this parameter are the same as the dwSpinCount parameter in Win32's InitializeCriticalSectionAndSpinCount. |

7

8      **HResults**

9      **See Common HResults**

10

11     **1.9.3. IHostAutoEvent**

12

13     interface IHostAutoEvent : IUnknown

14     {

15          HRESULT Wait([in] DWORD dwMilliseconds,

1       [in] DWORD option);

2       HRESULT Set();

3       }

4

5    *1.9.3.1.*       *Wait*

6       Waits until the event is owned or until the timeout expires.

7

| Parameter | Description |
|-----------|-------------|
| dwMilliseconds | [in] The number of milliseconds to wait until returning if the event doesn't become owned. |
| Option | [in] Values from WAIT_OPTIONS indicating actions the host 132 should take if this operation blocks. See description for WAIT_OPTIONS above. |

8

9    **HResults**

10    **See Common HResults**

11        **HOST_E_DEADLOCK**.  The host 132 detected a deadlock during

12    the wait and chose this lock as a deadlock victim.

13        **HOST_E_ABANDONED**.   The event handle was closed when

14    another thread was still waiting for it to be signaled.

15

16    *1.9.3.2.*       *Set*

17       The Set function sets the event to a signaled state.

1

2        **HResults**

3       **See Common HResults**

4

5     **1.9.4. IHostManualEvent**

6

```
7    interface IHostManualEvent : IUnknown
8    {
9       HRESULT Wait([in] DWORD dwMilliseconds,
10              [in] DWORD option);
11      HRESULT Reset();
12      HRESULT Set();
13   }
```
14

15   *1.9.4.1.     Wait*

16       Waits until the event is owned or until the timeout expires.

17

| Parameter | Description |
|---|---|
| dwMilliseconds | [in] The number of milliseconds to wait until returning if the event doesn't become owned. |
| Option | [in] Values from WAIT_OPTIONS indicating actions the host 132 should take if this operation blocks. See description for WAIT_OPTIONS above. |

18

19       **HResults**

1     **See Common HResults**

2         **HOST_E_DEADLOCK**.  The host 132 detected a deadlock during

3     the wait and chose this lock as a deadlock victim.

4         **HOST_E_ABANDONED**.   The event handle was closed when

5     another thread was still waiting for it to be signaled.

6

7 *1.9.4.2.*     *Reset*

8     The Reset function resets the event to a non-signaled state.

9

10     **HResults**

11     **See Common HResults**

12

13 *1.9.4.3.*     *Set*

14     The Set function sets the event to a signaled state.

15

16     **HResults**

17     **See Common HResults**

18

19 **1.9.5. IHostSemaphore**
20
21   interface IHostSemaphore : IUnknown
22   {
23     HRESULT Wait([in] DWORD dwMilliseconds,
24         [in] DWORD option);
25     HRESULT ReleaseSemaphore([in] LONG lReleaseCount,
26         [out] LONG *lpPreviousCount);
27   }
28

1     *1.9.5.1.*     *Wait*

2     Waits until the semaphore is owned or until the timeout expires. The

3   semaphore becomes owned when its count in non-zero.

4

| Parameter | Description |
|---|---|
| dwMilliseconds | [in] The number of milliseconds to wait until returning if the semaphore doesn't become owned. |
| Option | [in] Values from WAIT_OPTIONS indicating actions the host 132 should take if this operation blocks. See description for WAIT_OPTIONS above. |

5

6      **HResults**

7      **See <u>Common HResults</u>**

8         **HOST_E_DEADLOCK.** The host 132 detected a deadlock during

9   the wait and chose this lock as a deadlock victim.

10

11     *1.9.5.2.*     *ReleaseSemaphore*

12     ReleaseSemahore increases the semaphore's count by the specified amount.

13

| Parameter | Description |
|---|---|
| lReleaseCount | [in] As defined in Win32's ReleaseSemaphore |

| lpPreviousCount | [in] As defined in Win32's ReleaseSemaphore |
|---|---|

1

2  **HResults**

3  **See Common HResults**

4

5  **1.9.6. IRuntimeSyncManager**

6

7  interface IRuntimeSyncManager : IUnknown
8  {
9     HRESULT GetMonitorOwner ([in] SIZE_T Cookie,
10               [out] IHostTask **ppOwnerHostTask);
11
12    HRESULT CreateRWLockOwnerIterator([in] SIZE_T Cookie, [out]
13 SIZE_T *pIterator);
14    HRESULT GetRWLockOwnerNext([in] SIZE_T Iterator, [out] IHostTask
15 **ppOwnerHostTask);
16    HRESULT DeleteRWLockOwnerIterator([in] SIZE_T Iterator);
17 }
18

19  *1.9.6.1.     GetMonitorOwner*

20  GetMonitorOwner returns the task which owns the monitor identified by

21  the cookie. This is the task free to execute under this monitor. Other tasks may

22  block when attempting to acquire this monitor. This method can be used by the

23  host 132 as part of a deadlock detection scheme. The cookie is associated with the

24  monitor when it is created using IHostSyncManager::CreateMonitorEvent.

25

26  GetMonitorOwner always returns immediately and may be called anytime

27  after IHostSyncManager::CreateMonitorEvent – hosts do not have to wait until

28  there is a task waiting on the event. A call to release the event underlying the

1     monitor may block (but won't deadlock) if a call to GetMonitorOwner is in affect

2     on the cookie associated with that monitor.

3

4         The IHostTask returned is *AddRef*'d to prevent this task from exiting while

5     the pointer is held by the host 132. The host 132 should *Release* this pointer to

6     decrement the reference count when finished.

7

| Parameter | Description |
|---|---|
| Cookie | [in] The cookie associated with the monitor. The host 132 is responsible for ensuring that this cookie maps to an active monitor. That is, the monitor has not been freed. |
| ppOwnerHostTask | [out] A pointer to the host 132 task that currently owns the monitor. NULL is returned if no task has ownership. |

8

9        **HResults**

10       **See <u>Common HResults</u>**

11       **E_INVALIDARG.** An invalid cookie was passed in.

12

1    *1.9.6.2.       CreateRWLockOwnerIterator*

2          This method creates an iterator that hosts can use to determine the set of

3    tasks that are waiting on a reader/writer lock.  Hosts may call this method, and the

4    equivalent "Next" and "Delete" methods during deadlock detection.  A call to

5    release the event underlying the reader/writer lock may block (but won't deadlock)

6    if an iteration is in affect on the cookie associated with that lock.

7

| Parameter | Description |
|-----------|-------------|
| Cookie | [in] The cookie associated with the reader/writer lock.  The host 132 is responsible for ensuring that this cookie maps to an active lock.  That is, the lock has not been freed. |
| pIterator | [out] An iterator that can be passed to GetRWLockOwnerNext and DeleteRWLockOwnerIterator. |

8

9          **HResults**

10         **See <u>Common HResults</u>**

11         **E_INVALIDARG.**  An invalid cookie was passed in.

12         **HOST_E_INVALIDOPERATION.** This call was made on a thread that is

13    currently running managed code and therefore may interfere with the runtime

14    130's attempt to prepare for garbage collection.  CreateRWLockOwnerIterator

15    should only be called on threads that are not currently running managed code.

16

1     *1.9.6.3.*      ***GetRWLockOwnerNext***

2     Returns the next task that is blocked on this reader/writer lock. If no tasks

3     are blocked, NULL is returned. At this point the iteration is over and the host 132

4     should delete it using DeleteRWOwnerIterator.

5

6     The IHostTask returned is *AddRef*'d to prevent this task from exiting while

7     the pointer is held by the host 132. The host 132 should *Release* this pointer to

8     decrement the reference count when finished.

9

| Parameter | Description |
|---|---|
| Iterator | [in] The iterator created with CreateRWLockOwnerIterator. |
| ppOwnerHostTask | [out] A pointer to the next host task that is currently waiting on the lock. NULL is returned when no more tasks are waiting. |

10

11     **HResults**

12     **See <u>Common HResults</u>**

13     **E_INVALIDARG.** An invalid interator was passed in.

14

15     *1.9.6.4.*      ***DeleteRWLockOwnerIterator***

16     Deletes an iterator created with CreateRWLockOwnerIterator.

17

| Parameter | Description |
|---|---|

| Iterator | [in] The iterator created with CreateRWLockOwnerIterator. |
|---|---|

1

2      **HResults**

3      **See <u>Common HResults</u>**

4      **E_INVALIDARG.** An invalid iterator was passed in.

5

6    1.10. Assembly Loading Abstraction

7          The assembly loading abstraction consists of interfaces that allow hosts to

8    customize the assembly loading process.  Specifically, hosts can supply a list of

9    assemblies that should be loaded domain-neutral and customize the way

10   assemblies are located and loaded.   The interfaces in the Assembly Loading

11   Abstraction are:

12     • **<u>IHostAssemblyManager</u>**. The runtime 130 asks for this top level interface

13         through IHostControl::GetHostManager when the runtime 130 is

14         initialized.  If an implementation of this interface is provided, it is assumed

15         that the host 132 wishes to control some aspect of the assembly binding

16         process.  IHostAssemblyManager contains methods for the host 132 to

17         provide the list of assemblies that should be loaded domain-neutral, the list

18         of assemblies to which runtime 130 should bind, and to supply an

19         implementation of IHostAssemblyStore through which the host 132 can

20         implement their own custom assembly resolution process.

21     • **<u>IHostAssemblyStore</u>**. To load an assembly from somewhere other than the

22         file system, a host 132 typically catches an AssemblyResolve event on

23         System.AppDomain and provides a byte array containing the assembly's

1    bits. An implementation of IHostAssemblyStore provides additional host-

2    specific stores from which it can bind.  If an IHostAssemblyStore is

3    provided, runtime 130 will call back to the host 132 through this interface

4    when binding to an assembly.  The host 132 is free to load the assembly

5    from anywhere it chooses and with whatever rules it deems appropriate.  In

6    essence, hosts can use IHostAssemblyStore to completely "bypass"

7    Runtime 130 if so desired.

8    • A instance of the <u>AssemblyBindInfo</u> structure is passed to define the

9    assembly each time a binding request occurs to the custom assembly store.

10   This structure describes the identity of the assembly being requested along

11   with information about whether there is any version policy present on the

12   machine that might affect the bind.

13   • **<u>IRuntimeAssemblyReferenceList</u>**.

14

15   **1.10.1. IHostAssemblyManager**
16
17   interface IHostAssemblyManager: IUnknown
18   {
19     HRESULT GetDomainNeutralAssemblies([out]
20   IRuntimeAssemblyReferenceList** ppReferenceList);
21     HRESULT GetNonHostStoreAssemblies([out]
22   IRuntimeAssemblyReferenceList** ppReferenceList);
23      HRESULT GetAssemblyStore([out] IHostAssemblyStore
24   **ppAssemblyStore);
25      HRESULT GetHostApplicationPolicy([in] DWORD dwPolicy,
26       [in] DWORD dwAppDomainId,
27       [in, out] DWORD *pcbBufferSize,
28       [out, size_is(*pcbBufferSize)] BYTE  *pbBuffer
29       );
30   }
31

1    *1.10.1.1.    GetDomainNeutralAssemblies*

2          IHostAssemblyManager::GetDomainNeutralAssemblies          is          the

3    implementation of the requirement to allow a host 132 to specify the set of

4    assemblies to load domain-neutral at a finer granularity than we had in V1.  This

5    method returns an exact list of assemblies to be loaded domain neutral.  Hosts can

6    return a NULL interface pointer to indicate no list is specified.  In this case, the

7    STARTUP_LOADER_OPTIMIZATION_*          settings          passed          to

8    CorBindToRuntimeEx and/or the AppDomainSetup.LoaderOptimzation property

9    passed to AppDomain.CreateDomain will determine which assemblies are loaded

10   domain-neutral.

11

12         The runtime 130 calls GetDomainNeutralAssemblies when the runtime 130

13   is initialized – in this implementation, it is not called again.  This list of assemblies

14   should form a full closure – if an assembly is included in the list, all assemblies it

15   references should also be in the list.  If a full closure is not specified, the runtime

16   130 will throw a FileLoadException the first time it tries to resolve a reference

17   from an assembly in the list to an assembly that is not in the list.

18

19         The assemblies returned from GetDomainNeutralAssemblies augment

20   those          selected          by          the          loader          optimization          setting

21   (STARTUP_LOADER_OPTIMIZATION_*) the host 132 may have passed to

22   CorBindToRuntimeEx.  For example:

23   •     If     this     method     returns     a     null     list,     the

24         STARTUP_LOADER_OPTIMIZATION_*     flags     define     the     set     of

25         assemblies that will be loaded domain neutral.

1        If the host 132 passes

2        STARTUP_LOADER_OPTIMIZATION_SINGLE_DOMAIN (load no

3        assemblies domain neutral), then only the assemblies returned from

4        GetDomainNeutralAssembles will be loaded domain neutral.

5  •    If the host 132 passes

6        STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN (load all

7        assemblies domain neutral), then all assemblies will be loaded domain

8        neutral, regardless of what is returned from GetDomainNeutralAssembles.

9  •    If the host 132 passes

10       STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN_HOST (load

11       only strong named assemblies domain neutral), then all strong named

12       assemblies plus those returned from GetDomainNeutralAssembles will be

13       loaded domain neutral.

14

15       The same rules apply to the AppDomainSetup.LoaderOptimization option

16  that can be passed to AppDomain.CreateDomain.

17

| Parameter | Description |
|---|---|
| ppReferenceList | [out] interface pointer to IRuntimeAssemblyReferenceList containing references to assemblies the host 132 wishes to load domain neutral |

18

19       **HResults**

20       **See <u>Common HResults</u>**

21

1     *1.10.1.2.*       ***GetNonHostStoreAssemblies***

2          GetNonHostStoreAssemblies returns a list of assemblies that the host 132

3   wants the runtime 130 to bind to. Assemblies not included in the list are meant to

4   be loaded by the host 132 (see IHostAssemblyStore below). The runtime 130 calls

5   GetNonHostStoreAssemblies when the runtime 130 is initialized.

6

7          If GetNonHostStoreAssemblies returns a NULL interface pointer, the

8   runtime 130 will bind to all assemblies. The process generally works as follows:

9   •       Each time the runtime 130 is asked to resolve an assembly reference, it will

10           consult the list of assemblies via the interface pointer returned in this

11           method.

12   •       If the referenced assembly is in the list, the runtime 130 will bind as

13           normal.

14   •       If the referenced assembly is NOT in the list AND the host 132 has

15           provided an implementation of IHostAssemblyStore (returned through

16           IHostAssemblyManager::GetAssemblyStore), the runtime 130 will call

17           IHostAssemblyStore::ProvideAssembly to allow the host 132 to supply the

18           assembly.

19   •       If the referenced assembly is NOT in the list and the host 132 has NOT

20           provided an implementation of IHostAssemblyStore, the runtime 130 will

21           fail the bind.

22

| Parameter | Description |
|---|---|
| ppReferenceList | [out] interface pointer to IRuntimeAssemblyReferenceList containing references |

| | |
|---|---|
| | to assemblies the host 132 wishes to be loaded by the runtime 130 from outside the host 132 provided store (most often the GAC) |

1

2      **HResults**

3      See **Common HResults**

4      **E_OUTOFMEMORY**. Not enough memory available to create the list of

5  assemblies.

6

7  *1.10.1.3.      GetAssemblyStore*

8      GetAssemblyStore returns an interface pointer of type IHostAssemblyStore

9  to a host-implemented container of assemblies.  IHostAssemblyStore provides

10  methods that allow hosts to bind to assemblies and modules independent of

11  runtime 130.  Hosts typically provide assembly stores to allow assemblies to be

12  loaded from formats other than the file system.  For example, SQL Server will

13  implement an assembly store to load assemblies from the database, while Avalon

14  may provide an implementation that loads assemblies out of their application

15  container files.

16

| Parameter | Description |
|---|---|
| ppAssemblyS tore | [out]  A  pointer  to  a  host's  implementation  of IHostAssemblyStore.  Return NULL If the host 132 doesn't implement a custom assembly store. |

17

1      **HResults**

2      **See <u>Common HResults</u>**

3      **E_NOINTERFACE.** The host 132 does not provide an implementation of

4 IHostAssemblyStore.

5

6      **1.10.2. IHostAssemblyStore**

7      As described, the IHostAssemblyStore interface gives the host 132 a way to

8 efficiently load assemblies from a host-specific store based on assembly identity.

9 Hosts load assemblies and modules by providing the runtime 130 with an IStream

10 which points directly to the bytes. By providing an implementation of

11 IHostAssemblyStore, the host 132 specifies its intent to resolve all assemblies *not*

12 referenced by the IRuntimeAssemblyReferenceList returned from

13 IHostAssemblyManager::GetNonHostStoreAssemblies. This allows hosts to

14 control binding to user assemblies, but still have the runtime 130 bind to the other

15 assemblies, if desired.

16

17      The runtime 130 determines if the host 132 has implemented

18 IHostAssemblyStore by calling IHostAssemblyManager::GetAssemblyStore at

19 startup.

20      Fig. 8 shows how IHostAssemblyStore fits into the assembly loading

21 architecture.
22
23    typedef struct _AssemblyBindInfo
24    {
25      DWORD                dwAppDomainId;
26      LPCWSTR           lpReferencedIdentity;
27      LPCWSTR           lpPostPolicyIdentity;
28      DWORD                ePolicyLevel;

```
1        }AssemblyBindInfo;
2
3        typedef struct _ModuleBindInfo
4        {
5          DWORD              dwAppDomainId;
6          LPCWSTR            lpAssemblyIdentity;
7          LPCWSTR            lpModuleName
8        }ModuleBindInfo;
9              interface IHostAssemblyStore: IUnknown
10             {
11                HRESULT ProvideAssembly
12                   (
13                 [in] AssemblyBindInfo *pBindInfo,
14                 [out] BYTE        *pbModuleId,
15                 [out] DWORD        *pcbModuleId,
16                 [out] IUnknown     **ppUnkEvidence,
17                 [out] IStream      **ppStmAssemblyImage,
18                 [out] IStream      **ppStmPDB);
19
20                HRESULT ProvideModule
21                   (
22                 [in] ModuleBindInfo *pBindInfo,
23                 [out] BYTE        *pbModuleId,
24                 [out] DWORD        *pcbModuleId,
25                 [out] IStream      **ppStmModuleImage,
26                 [out] IStream      **ppStmPDB);
27             }.
28
```

29 *1.10.2.1.    ProvideAssembly*

30      This interface is called to resolve all assemblies NOT referenced by the

31 IRuntimeAssemblyReferenceList                returned                from

32 IHostAssemblyManager::GetNonHostStoreAssemblies.    Information about the

33 assembly including identity and pre and post policy references are supplied to the

34 host 132 via an instance of <u>AssemblyBindInfo</u> (the pBindInfo parameter).

35

1    Assemblies are returned from the host 132 to Runtime 130 as an IStream*.

2    In addition to the stream itself, the host 132 also returns an identifier that runtime

3    130 can use to uniquely identify that stream. The id is completely host-specified –

4    the contents of the id itself have no meaning to the runtime 130. This id is used

5    within the runtime 130 as the identity of the mapped stream. Each time the host

6    132 returns a stream with the same id, the runtime 130 consults a table to see if the

7    contents of that stream have already been mapped. If so, the existing copy is used

8    instead of remapping the stream. The id's should be unique within the lifetime of

9    a given process.

10

| Parameter | Description |
|---|---|
| pBindInfo | [in] An instance of AssemblyBindInfo through which the host 132 determines various characteristics of the bind including which assembly to resolve and whether any version policy is present that would affect the original reference. |
| pbModuleID | [out] An identifier used to uniquely identify this stream. If the same id is returned from multiple calls to ProvideAssembly, the stream representing that id will only be mapped once. These module ids should be unique from those used in the ProvideModule API below |

| | |
|---|---|
| | because they occupy the same logical namespace. |
| pcbModuleID | [in, out] The size of the buffer (pbModuleID) used to hold the id. An initial value is passed in with the call. If this buffer is not big enough, the host 132 will return the required size back through pcbModuleID and return ERROR_INSUFFICIENT_BUFFER from ProvideAssembly. |
| ppUnkEvidence | [out] The evidence the host 132 wants associated with the loaded assembly. |
| ppStmAssemblyImage | [out] An IStream that contains the PE image to be loaded. NULL if the assembly couldn't be found. |
| ppStmPDB | [out] An IStream that contains the pdb (debug information). NULL if the pdb file cannot be loaded |

1

2      **HResults**

3      **See Common HResults**

4      **ERROR_FILE_NOT_FOUND.**   The requested assembly could not be

5   found.

1      **ERROR_INSUFFICIENT_BUFFER.** The size of the buffer passed in

2      through pcbAssemblyID is not big enough to hold the id the host 132 wishes to

3      return.

4

5      *1.10.2.2.      ProvideModule*

6          Runtime 130 calls this method to resolve a module within an assembly.

7

| Parameter | Description |
|---|---|
| pBindInfo | [in]      An      instance      of ModuleBindInfo      describing      the requested      module's      AppDomain, assembly and module name. |
| pbModuleId | [out]   An   identifier   used   to uniquely  identify  this  stream.   If the same id is returned from multiple calls to ProvideModule,  the  stream  representing that id will only be mapped once.  The module id used here should be unique from  those  used  in  ProvideAssembly above  because  these  identifiers  occupy the same logical namespace |
| pcbModuleID | [in, out] The  size  of  the  buffer (pbModuleID) used to hold the id.  An initial value is passed in with the call.  If |

| | |
|---|---|
| | this buffer is not big enough, the host 132 will return the required size back through pcbModuleID and return ERROR_INSUFFICIENT_BUFFER from ProvideModule. These id's should be unique process wide and should not collide with id's returned from ProvideAssembly. |
| ppStmModuleImage | **[out] An IStream that contains the PE image to be loaded. NULL if the module couldn't be found.** |
| **ppstmModuleDebugInfo** | **[out] An IStream containing the pdb "file" for this module. NULL if the debug info couldn't be found or if debug info is not requested.** |

1

2      **HResults**

3      **See <u>Common HResults</u>**

4      **ERROR_FILE_NOT_FOUND.** The requested assembly could not be

5    found.

6      **ERROR_INSUFFICIENT_BUFFER.** The size of the buffer passed in

7    through pcbModuleID is not big enough to hold the id the host 132 wishes to

8    return.

9

1    **1.10.3. IRuntimeAssemblyReferenceList**

2        The host 132 communicates information about assemblies, such as the list

3    to load domain-neutral or the list loaded by the runtime 130 (not from the host 132

4    store) by creating a list of these assembly references, accessed by an instance of

5    IRuntimeAssemblyReferenceList.          This     instance     is     created     via

6    IRuntimeAssemblyIdentityManager::GetRuntimeAssemblyReferenceList.

7

8        interface IRuntimeAssemblyReferenceList : IUnknown

9        {

10         HRESULT   IsStringAssemblyReferenceInList(

11           [in]   LPCWSTR   pwzAssemblyName

12         );

13

14         HRESULT   IsAssemblyReferenceInList(

15           // pName could be IAssemblyName or IReferenceIdentity

16           [in]   IUnknown *pName

17         );

18        }

19

20    *1.10.3.1.      IsStringAssemblyReferenceInList*

21        Given an assembly name, determine if the name is in the assembly

22    reference list.

23

| Parameter | Description |
|---|---|
| pwszAssemblyName | [in] Name of the assembly to search for in list |

24        **HResults**

25                **S_OK**        The string or reference is found in list.

26                **S_FALSE** The string or reference is not found in list.

27                **E_FAIL** Internal failure

1

2 *1.10.3.2.      Is AssemblyReferenceInList*

3

| Parameter | Description |
|-----------|-------------|
| pName | [in] IUnknown pointer to either IAssemblyName or IReferenceIdentity for assembly to search for in list |

4        **HResults**

5                         S_OK The string or reference is found in list.

6                         S_FALSE The string or reference is not found in list.

7                         E_FAIL Internal failure

8

9  **1.10.4. IRuntimeHostBindingPolicyManager**

10        This interface allows the host 132 to communicate changes in policy

11  information for a particular assembly and to evaluate the currently policy.  The

12  host 132 supplies the source and target assembly identities with current policy and

13  the runtime 130 will return the new application of policy between them.
14
15     typedef enum
16     {
17        ePolicyLevelNone = 0x0,
18        ePolicyLevelRetargetable = 0x1,
19        ePolicyUnifiedToRuntime = 0x2,
20        ePolicyLevelApp = 0x4,
21        ePolicyLevelPublisher = 0x8,
22        ePolicyLevelHost = 0x10,
23        ePolicyLevelAdmin = 0x20
24     } EBindPolicyLevels;
25
26     interface IRuntimeHostBindingPolicyManager : IUnknown

```
1      {
2          HRESULT   ModifyApplicationPolicy(
3              [in]   LPCWSTR        pwzSourceAssemblyIdentity,
4              [in]   LPCWSTR        pwzTargetAssemblyIdentity,
5              [in]   BYTE           *pbApplicationPolicy,
6              [in]   DWORD          cbAppPolicySize,
7              [in]   DWORD          dwPolicyModifyFlags,
8              [out, size_is(*pcbNewAppPolicySize)]BYTE
9          *pbNewApplicationPolicy,
10             [in, out]  DWORD    *pcbNewAppPolicySize
11         );
12
13         HRESULT   EvaluatePolicy(
14             [in]   LPCWSTR         pwzReferenceIdentity,
15             [in]   BYTE            *pbApplicationPolicy,
16             [in]   DWORD           cbAppPolicySize,
17             [out, size_is(*pcchPostPolicyReferenceIdentity)]   LPWSTR
18         pwzPostPolicyReferenceIdentity,
19             [in, out] DWORD         *pcchPostPolicyReferenceIdentity,
20             [out]   DWORD           *pdwPoliciesApplied
21         );
22      };
23
```

### 1.10.4.1.     *ModifyApplicationPolicy*

Given a new binding redirect, modify the old binding policy, and return a new copy.

| Parameter | Description |
|---|---|
| pwszSourceAssemblyIdentity | [in] Old version of assembly |
| pwszTargetAssemblyIdentity | [in] New version of assembly |
| pbApplicationPolicy | [in] Old policy (opaque data) |
| cbAppPolicySize | [in] Size of the old policy |
| dwPolicyFlags | [in]   EHostBindingPolicyFlags values, controlling redirection |

| pbNewApplicationPolicy | [out] New application policy (opaque data) |
|---|---|
| pcbNewAppPolicySize | [in, out] Size of new policy buffer |

1

2    In one implementation, this API is called twice: once to ascertain the

3    necessary size of the new policy buffer, by passing NULL for the new policy

4    buffer.    The method will return the necessary buffer size value in

5    pcbNewAppPolicySize.  The second call should pass this same value, and pass a

6    correctly allocated and sized buffer in pbNewApplicationPolicy.

7

8    **HResults**

9        **S_OK**        SUCCESS

10        **E_INVALIDARG**            if   pwzSourceAssemblyIdentity   or

11    pwzTargetAssemblyIdentity is NULL.

12        **HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFE**

13    **R)** if input buffer is too small.

14

15    *1.10.4.2.    EvaluatePolicy*

16        This method evaluates policy on behalf of the host 132.  The intent of these

17    API's is to allow the host 132 to influence policy to enforce host-specific

18    versioning and in-place assembly updates, but to keep the policy engine itself

19    within the runtime 130, maintaining long term policy consistency.

| Parameter | Description |
|---|---|

| | |
|---|---|
| pwszReferenceIdentity | [in] pre-policy assembly reference |
| pbApplicationPolicy | [in] policy data (opaque) |
| cbAppPolicySize | [in] size of policy data buffer |
| pwszPostPolicyReferenceIdentity | [out] post-policy assembly reference |
| pcchPostPolicyReferenceIdentity | [in,out] size of post-policy assembly reference buffer |
| pdwPoliciesApplied | [out] policies applied. Combination of values or'd from EBindPolicyLevels enum |

1

### 1.10.5. IRuntimeAssemblyIdentityManager

```
interface IRuntimeAssemblyIdentityManager : IUnknown
{
   HRESULT  GetRuntimeAssemblyReferenceList(
      [in]   LPCWSTR  *ppwzAssemblyReferences,
      [in]   DWORD   dwNumOfReferences,
      [out]  IRuntimeAssemblyReferenceList   **ppReferenceList
   );
   HRESULT GetTextualIdentityFromFile(
      [in]   LPCWSTR   pwzFilePath,
      [in]   DWORD    dwFlags,
      [out, size_is(*pcchBufferSize)]   LPWSTR pwzBuffer,
      [in, out]  DWORD   *pcchBufferSize
   );
   HRESULT GetTextualIdentityFromStream(
      [in]   IStream   *pStream,
      [in]   DWORD    dwFlags,
      [out, size_is(*pcchBufferSize)]   LPWSTR pwzBuffer,
      [in, out]  DWORD    *pcchBufferSize
   );
```

```
1    HRESULT GetReferencedAssembliesFromFile(
2        [in]   LPCWSTR              pwzFilePath,
3        [in]   DWORD                dwFlags,
4        [in]   IRuntimeAssemblyReferenceList  *pExcludeAssembliesList,
5        [out]  IRuntimeReferenceAssemblyEnum  **ppReferenceEnum
6    );
7    HRESULT GetReferencedAssembliesFromStream(
8        [in]   IStream              *pStream,
9        [in]   DWORD                dwFlags,
10       [in]   IRuntimeAssemblyReferenceList  *pExcludeAssembliesList,
11       [out]  IRuntimeReferenceAssemblyEnum  **ppReferenceEnum
12   );
13   HRESULT GetProbingAssembliesFromReference(
14       [in]   DWORD            dwMachineType,
15       [in]   DWORD            dwFlags,
16       [in]   LPCWSTR          pwzReferenceIdentity,
17       [out]  IRuntimeProbingAssemblyEnum    **ppProbingAssemblyEnum
18   );
19   };
20
```

21   *1.10.5.1.      GetRuntimeAssemblyReferenceList*

22       This API returns an instance of IRuntimeAssemblyReferenceList from a

23   text list of partial assembly identities.  This is the bridge between host / human

24   readable partial identities and the unique opaque representation used internally for

25   policy evaluation and application.

26

| Parameter | Description |
|---|---|
| ppwszAssemblyReferences | **[in] Array of NULL terminated strings, in forms of "name, property=value, ..."** |
| dwNumOfReferences | **[in] number of strings in array above** |
| **ppReferenceList** | **[out] returned interface pointer** |

| | to IRuntimeAssemblyReferenceList |
|---|---|

1

2      **HResults**

3      See <u>**Common HResults**</u>

4

5   *1.10.5.2.      GetTextualIdentityFromFile*

6      This API returns the opaque canonical assembly identity, used internally

7   for policy application and evaluation, from a given assembly file.

8

| Parameter | Description |
|---|---|
| pwszFilePath | [in] path to file to be evaluated |
| dwFlags | [in] |
| pwszBuffer | [out]        assembly's        identity (opaque data) |
| pcchBufferSize | [in,   out]   buffer   size   of pwszBuffer |

9

10      This API may be called twice, once to ascertain the necessary size of

11   pwszBuffer, by passing NULL for that and receiving the necessary buffer size

12   back in pcchBufferSize.   The second call would pass a correctly allocated

13   pwszBuffer, and supply that size in pcchBufferSize.   Upon return pwszBuffer

14   would be filled in with the opaque textual identity.

15

16      **HResults**

1      **S_OK** SUCCESS

2      **E_INVALIDARG** If pwzFilePath is NULL.

3   •   **HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER)**

4      If the supplied buffer is too small

5   •   **E_FAIL** Other

6

7   *1.10.5.3.*      *GetTextualIdentityFromStream*

8      This API returns the opaque canonical assembly identity, used internally

9   for policy application and evaluation, from a given assembly IStream.

10

| Parameter | Description |
|---|---|
| pStream | [in] assembly stream to be evaluated |
| dwFlags | [in] |
| pwszBuffer | [out] assembly's identity (opaque data) |
| pcchBufferSize | [in, out] buffer size of pwszBuffer |

11

12      This API may be called twice, once to ascertain the necessary size of

13   pwszBuffer, by passing NULL for that and receiving the necessary buffer size

14   back in pcchBufferSize.   The second call would pass a correctly allocated

15   pwszBuffer, and supply that size in pcchBufferSize.   Upon return pwszBuffer

16   would be filled in with the opaque textual identity.

1

2    **HResults**

3    •    **S_OK** SUCCESS

4    •    **E_INVALIDARG** If pwzFilePath is NULL.

5    •    **HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER)**    If

6    the supplied buffer is too small

7    •    **E_FAIL**    Other

8

9    *1.10.5.4.    GetReferencedAssembliesFromFile*

10   This API returns post-policy references for the assemblies referenced from

11   this file. The caller may choose to exclude a known set of assembly references

12   from    what    is    returned.    This    set    is    defined    by    an    instance    of

13   IRuntimeAssemblyReferenceList.

14

| Parameter | Description |
|---|---|
| pwszFilePath | [in] path to assembly file to be evaluated |
| dwFlags | [in] |
| pExcludeAssembliesList | [in] interface pointer to list of assembly references to exclude |
| pwszBuffer | [out] assembly's identity (opaque data) |
| pcchBufferSize | [in, out] buffer size of pwszBuffer |

1

2      **HResults**

3      See <u>Common HResults</u>

4

5    *1.10.5.5.*       *GetReferencedAssembliesFromStream*

6      This API returns post-policy references for the assemblies referenced from

7  this file. The caller may choose to exclude a known set of assembly references

8  from what is returned. This set is defined by an instance of

9  IRuntimeAssemblyReferenceList.

10

| Parameter | Description |
|---|---|
| pStream | [in] IStream pointer to assembly to be evaluated |
| dwFlags | [in] |
| pExcludeAssembliesList | [in] interface pointer to list of assembly references to exclude |
| pwszBuffer | [out] assembly's identity (opaque data) |
| pcchBufferSize | [in, out] buffer size of pwszBuffer |

11

12      **HResults**

13      See <u>Common HResults</u>

14

1    *1.10.5.6.        GetProbingAssembliesFromReference*

2        Based on the canonical textual identity of an assembly, returns an enum

3    that may produce identities for processor architecture specific, MSIL only, or

4    identity with no architecture information.

5

| Parameter | Description |
|---|---|
| dwMachineType | [in] valid processor architecture as defined in winnt.h |
| dwFlags | [in] |
| pwszReferenceIdentity | [in] opaque textual assembly identity (possibly returned from GetTextualIdentityFromStream or GetTextualIdentityFromFile) |
| ppProbingAssemblyEnum | [out] instance of enumerator interface containing all runtime 130 probing assemblies |

6

7    **1.10.6. IRuntimeProbingAssemblyEnum**

8        An instance of this interface is returned from method such as

9    IRuntimeAssemblyIdentityManager::GetProbingAssembliesFromReference.    It

10   allows the host 132 to acquire the probing identities of an assembly using the

11   canonical identity (internal representation of the runtime 130) without being

12   required to understand or form that identity.

13

14       interface IRuntimeProbingAssemblyEnum:IUnknown

15       {

```
1    HRESULT Get(
2        [in]      DWORD   dwIndex,
3        [out, size_is(*pcchBufferSize)]   LPWSTR  pwzBuffer,
4        [in, out]  DWORD   *pcchBufferSize
5        );
6    };
7
```

8    *1.10.6.1.    Get*

| Parameter | Description |
|---|---|
| dwIndex | [in] index of assembly identity to return |
| pwszBuffer | [in] |
| pcchBufferSize | [in, out] opaque textual assembly identity |

9

10    The identity at index 0 will be the processor architecture specific identity.

11    The identity at index 1 is the MSIL architecture-neutral identity, The identity at

12    index 2 will not contain architecture information.

13

14    This API may be called twice, once supplying NULL for pwszBuffer and

15    upon return pcchBufferSize contains the necessary size of pwszBuffer.  On the

16    second call, pass this value for pcchBufferSize and a correctly allocated

17    pwszBuffer.  Upon return from this second call, pwszBuffer will be correctly filled

18    with the canonical assembly identity.

19    **HResults**

20        **S_OK** SUCCESS

1    **HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFE**

2    **R)**   If the supplied buffer is too small

3    **HRESULT_FROM_WIN32(ERROR_NORE_MORE_ITEMS)**

4    If end of enum

5

6    **1.10.7. IRuntimeReferenceAssemblyEnum**

7    An instance of this interface is returned from method such as

8    IRuntimeAssemblyIdentityManager::GetReferencedAssembliesFromFile.    It

9    allows the host 132 to manipulate the set of assemblies referenced by a file or

10   stream using the canonical identities (internal representation of the runtime 130)

11   without being required to understand or form those identities.
12
13   interface IRuntimeReferenceAssemblyEnum : IUnknown
14   {
15     HRESULT Get(
16        [in]      DWORD  dwIndex,
17        [out, size_is(*pcchBufferSize)]   LPWSTR  pwzBuffer,
18        [in, out]  DWORD   *pcchBufferSize
19     );
20   };
21

22   *1.10.7.1.    Get*

| Parameter | Description |
|---|---|
| dwIndex | [in] index of assembly identity to return |
| pwszBuffer | [in] |
| pcchBufferSize | [in, out] opaque textual assembly identity |

1    This API may be called twice, once supplying NULL for pwszBuffer and

2    upon return pcchBufferSize contains the necessary size of pwszBuffer.  On the

3    second call, pass this value for pcchBufferSize and a correctly allocated

4    pwszBuffer.  Upon return from this second call, pwszBuffer will be correctly filled

5    with the canonical assembly identity, without processor architecture.

6

7    **HResults**

8    **S_OK** SUCCESS

9    **HRESULT_FROM_WIN32(ERROR_INSUFFICIENT_BUFFER)**

10    If the supplied buffer is too small

11    **HRESULT_FROM_WIN32(ERROR_NORE_MORE_ITEMS)**

12    If end of enum.

13

14    **1.11. Security Context and Impersonation**

15    Hosts 132 may choose to control all framework and user code access to

16    thread tokens and to ensure complete security context information is passed across

17    async points or points of restricted context execution.

18

19    The actual context information unique to the host 132 is encapsulated in an

20    instance of an interface IHostSecurityContext.  This is opaque to the runtime 130

21    and will be captured and moved across threadpool worker item dispatch, finalizer

22    execution, and both module and class constructor execution.

23

24    **Interface Definitions**
25

```
1    typedef enum
2    {
3        eCurrentContext=0x00,
4        eRestrictedContext=0x01
5    } EContextType;
6
```

7    **1.11.1. IHostSecurityContext**

8

```
9    interface IHostSecurityContext : IUnknown
10   {
11       HRESULT Capture([out] IHostSecurityContext** ppClonedContext);
12   }
13
```

14   **1.11.2. IHostSecurityManager**

15

```
16   interface IHostSecurityManager : IUnknown
17   {
18       HRESULT ImpersonateLoggedOnUser([in] HANDLE hToken);
19       HRESULT RevertToSelf();
20       HRESULT OpenThreadToken([in] DWORD dwDesiredAccess,
21           [in] BOOL bOpenAsSelf, [out] HANDLE *phThreadToken);
22       HRESULT SetThreadToken([in] HANDLE hToken);
23       HRESULT GetSecurityContext([in] EContextType eContextType,
24           [out] IHostSecurityContext** ppSecurityContext);
25       HRESULT SetSecurityContext([in] EContextType eContextType,
26           [in] IHostSecurityContext* pSecurityContext);
27   }
```

28   **1.11.3. Impersonation**

29         Note, only API's actually affecting a thread are covered here. There are a

30   number of other API's such as *CreateRestrictedToken* and *LogonUser* which

31   produce tokens but do not affect a thread. Those tokens may then be passed to one

32   of the API's above.

33

34         The first 4 methods of IHostSecurityManager mirror those of the Win32

35   API, with the difference that the Win32 API versions allow passing of an arbitrary

1    thread handle, but the host 132 SecurityManager API's all act on the current

2    thread of execution.

3

4         In the case of OpenThreadToken, an actual OS token is returned and the

5    desired access level is achieved. User code may then call other API's including,

6    for example, the WIN32 API (e.g., DuplicateToken or GetTokenInformation)

7    passing this token. The other API's will expect real OS tokens to be passed in as

8    well, because these tokens may be coming from other API's such as Win32

9    *LogonUser*.

10

11   *1.11.3.1.       ImpersonateLoggedOnUser*

12

| Parameter | Description |
|-----------|-------------|
| hToken | [in] token representing the credentials of the user to impersonate. This may have been retrieved using LogonUser or via a Win32 API. |

13

14   *1.11.3.2.       RevertToSelf*

15        This function terminates the client impersonation and returns the original

16   thread token.

17

1    *1.11.3.3.      OpenThreadToken*

2

| Parameter | Description |
|---|---|
| dwDesiredAccess | [in] Specifies an access mask that specifies the requested types of access to the access token. These requested access types are reconciled against the token's discretionary access control list (DACL) to determine which accesses are granted or denied. See Win32 documentation for OpenThreadToken API. |
| bOpenAsSelf | [in] Indicates whether the access check is to be made against the calling thread token or against the process token. See Win32 documentation for OpenThreadToken API. |
| phThreadToken | [out] Pointer to the newly acquired thread token |

3

4    *1.11.3.4.      SetThreadToken*

| Parameter | Description |
|---|---|

| pAssemblyInfo | [in] pointer to AssemblyBindInfo instance containing Assembly information |
|---|---|
| pAssemblyId | [out] 64 bit process unique host 132 identifier for this assembly |

1

## 1.11.4. Security Context Flow

Managed thread context will be handled internally to the runtime 130 and passed across async points.

See ExecutionContext for internal details. These will include the compressed stack for CAS evaluation. Information unique to the host 132 is exposed by via **IHostSecurityContext,** captured by the runtime 130 at each point below and propagated to the receiving side. The runtime 130 will query the process-wide IHostSecurityManager to retrieve the IHostSecurityContext appropriate to the current thread of execution in the following cases:

- Finalizer Execution (on finalizer thread)
- Class and module constructor execution (.cctors and .mtors)
- Worker Thread Async Points (ThreadPool QueueUserWorkItem)
- Async I/O (I/O completion port servicing)

## 1.11.5. Enum: EContextType

The runtime 130 currently supports two context types, the *current* thread context, which is the context on the thread at the time the runtime 130 calls

1   GetSecurityContext on the host 132, and *restricted* context which is a lower

2   privilege context. The runtime does not allow a host 132 to differentiate restricted

3   contexts, i.e. it asks for eRestrictedContext to run finalizers, class and module

4   constructors.

5

| Parameter | Description |
|-----------|-------------|
| eCurrentContext | Context on thread at the time runtime 130 calls GetSecurityContext |
| eRestrictedContext | Lower privilege context placed on thread to run finalizers, class and module constructors |

6

7   *1.11.5.1.     GetSecurityContext*

8

| Parameter | Description |
|-----------|-------------|
| eContextType | [in] value from EContextType denoting what type of context to return |
| ppSecurityContext | [out] interface pointer to IHostSecurityContext |

9

1    *1.11.5.2.*        *SetSecurityContext*

2

| Parameter | Description |
|---|---|
| eContextType | [in] value from EContextType denoting what type of context runtime 130 is placing on host |
| pSecurityContext | [in] interface pointer to IHostSecurityContext of eContextType. |

3

4    *1.11.5.3.*        *Capture*

5

| Parameter | Description |
|---|---|
| ppClonedContext | [out] interface pointer to IHostSecurityContext. This is a clone of the context to capture. When this information is moved across an async point, it's lifetime is separated from the original queried context pointer, which may then be released. |

6

7    **1.11.6. Execution of Finalizers, Class, and Module Constructors**

8    Hosts 132 may disallow user code finalizers, but even if only fully trusted

9    .NET Framework code finalizers run, they may call back into arbitrary user code.

1   In addition, the runtime 130 may choose to run class or module constructors in the

2   host 132's restricted context. (NOTE: we do not provide multiple notions of

3   restricted contexts. The same restricted context used for finalizers is used for

4   these constructors.)

5

6       The runtime 130 protects the host 132 by doing the following:

7   •    Capture the *current* thread context by calling

8       IHostSecurityManager::GetSecurityContext(eCurrentContext,

9       &pCurrentContext)

10   •    Capture the *restricted* context by calling

11       IHostSecurityManager::GetSecurityContext(eRestrictedContext,

12       &pRestrictedContext)

13   •    Apply the *restricted* context by calling

14          o pRestrictedContext->Capture(&pClonedRestrictedContext)

15          o IHostSecurityManager::SetSecurityContext(eRestrictedContext,

16             pClonedRestrictedContext)

17   •    Run the finalizer, class or module constructor

18   •    Replace the original *current* context by calling

19       IHostSecurityManager::SetSecurityContext(eCurrentContext,

20       pCurrentContext)

1    The failure modes are similar to the finalizer thread, with one exception.

2 The runtime 130 are able to re-apply the original (defined as *current*) context to

3 the thread after class constructor execution. If not, subsequent code may fail when

4 running under the *restricted* context.

5

6 **1.11.7. Worker Thread Async Point**

7    When    hosted,    the    runtime    130    will    call

8 IHostSecurityManager::GetSecurityContext    within

9 Threadpool.QueueUserWorkItem at the same point it collects compressed stack

10 and other thread security context

11    There are two cases to consider. Either the runtime 130 or the host 132 can

12 supply the threadpool implementation.

13

14 • **Runtime supplied threadpool**

15    1. Within QueueUserWorkItem, delegate info (including the delegate itself

16       and additional security info) is gathered and used as the state information

17       supplied to the internal QueueUserWorkItem.

18    **2.** The runtime 130 calls

19       **IHostSecurityManager::GetSecurityContext(eCurrentContext,**

20       **&pCurrentContext)**

21          a.   to retrieve a host 132 security context interface pointer.

22          *b.* Calls pCurrentContext->Capture(&pCapturedContext)

23          c.   Places the captured (cloned) information in the delegate info.

1    3. When the actual worker thread dequeues and executes the work request,

2        this info is retrieved and the runtime 130 calls

3        IHostSecurityManager::SetSecurityContext(eCurrentContext,

4        pCapturedContext) prior to the function execution

5    • **Host 132 supplied threadpool**

6    **1.** Within ThreadPoolMgr::QueueUserWorkItem we query the host 132 for

7        *IHostThreadPoolManager*

8    **2.** The runtime 130 calls

9        o Calls *IHSM::GetSecurityContext(eCurrentContext,*

10        *&pCurrentContext)* to retrieve a host 132 security context interface

11        pointer.

12        o Calls *pCurrentContext->Capture(&pCapturedContext)*

13        o Places the captured (cloned) information in the delegate info.

14    **3.** The runtime 130 calls *IHTPM::QueueUserWorkItem,* passing the

15        QueueUserWorkItemCallback function pointer and delegate info as

16        function parameter.  This delegate info contains runtime 130 security

17        information including compressed stack and includes the context garnered

18        from the host 132 in item #2.

19    **4.** When the host 132 dequeues the work request, it calls into the runtime 130

20        on QueueUserWorkItemCallback, passing the delegate info. The runtime

21        130 uses information to apply the compressed stack to the new thread and

1      calls IHostSecurityManager::SetSecurityContext(eCurrentContext,

2      pCapturedContext) using the context unbundled from the delegate info.

3      **Async I/O**

4      The execution context is captured in runtime 130 internal class which is

5  wrapped and passed within the native overlapped structure.

6      For the host 132ing case, the runtime 130 will capture both the additional

7  managed security context information, and the host 132-specific context, using

8  IHostSecurityContext, within this internal managed instance.

9

10    1. The runtime 130 captures managed and host 132 context

11        a. Calls IHSM::GetSecurityContext(eCurrentContext,

12          `&pCurrentContext) to retrieve a host 132 security context interface

13          pointer.

14        b. Calls pCurrentContext->Capture(&pCapturedContext),

15        c. Places the captured (cloned) COM interface into the managed

16          instance within the native overlapped structure.

17    2. If the host 132 implements IHostIOCompletionManager, the runtime 130

18      calls its implementation of Bind, else it calls the runtime internal IO

19      completion routines.

20    3. In the non-hosted case, upon OS signaling async I/O completion, control

21      returns to an internal runtime 130 routine. In the hosted case, the host 132

22      alerts the runtime of I/O completion by firing the

1    IRuntimeIoCompletionManager::OnComplete.  Control then returns to the

2    same internal runtime 130 routine.

3    4.  Within this routine, the managed security context is reapplied. The runtime

4    130 will then call IHSM::SetSecurityContext(eCurrentContext,

5    pCurrentContext) to replace host 132 security context before further

6    managed code executes.

7    1.12. Runtime Configuration Interfaces

8    Interfaces 133 (fig. 1) provide various abstractions that hosts 132 can

9    implement to provide and monitor resources used by the runtime 130.  In this

10   implementation, runtime 130 queries for these interfaces at startup to determine

11   whether a host 132 supports a given abstraction.  This is done because whether a

12   particular abstraction is supported is central to how the runtime 130 executes.

13

14   Additionally, there are a set of configuration parameters (program data 138)

15   that a host 132  can set that are not required up front or that can change over time.

16   Examples include a host's policy for escalation resource failures, or to group a set

17   of related tasks for display in the debugger.  These parameters are set through

18   interfaces 134 that the host 132 obtains from the runtime 130.  These interfaces

19   include, for example:

20   •    IRuntimeControl

21   •    IRuntimePolicyManager

22   •    IRuntimeDebugManager

23   •    IRuntimeHostProtectionManager

24        IRuntimeGCManager

1       IRuntimeOnEventManager

2

3       This section of the spec describes the runtime 130 configuration interfaces

4   134.

5

6   **1.12.1. IRuntimeControl**

7       IRuntimeControl is the "starting point" for the remainder of the

8   configuration interfaces.  Hosts 132 obtain a pointer to an IRuntimeControl by

9   calling IRuntimeRuntimeHost::GetRuntimeControl.  From this interface hosts 132

10   can get pointers to other interfaces that allow various aspects of the runtime 130 to

11   be configured.

12

13       interface IRuntimeControl : IUnknown

14       {

15       HRESULT GetRuntimeManager([in] REFIID riid, [out] void **ppObject);

16       }

17

18   *1.12.1.1.      GetRuntimeManager*

19       Returns an interface pointer to one of the runtime 130 "managers" used to

20   configure the runtime 130.  The following interfaces can be returned from

21   GetRuntimeManager:

22   •       IRuntimeDebugManager (IID_IRuntimeDebugManager)

23   •       IRuntimePolicyManager (IID_IRuntimePolicyManager)

24   •       IRuntimeHostProtectionManager (IID_ IRuntimeHostProtectionManager)

25   •       IRuntimeOnEventManager (IID_IRuntimeOnEventManager)

1 • <u>IRuntimeGCManager</u> (IID_IRuntimeGCManager)

2 • IRuntimeAssemblyIdentityManager

3     (IID_IRuntimeAssemblyIdentityManager)

4 • <u>IRuntimeHostPolicyManager</u> (IID_IRuntimeHostPolicyManager)

5

| Parameter | Description |
|-----------|-------------|
| Riid | [in] The IID for the manager to return. Current valid values are listed above. |
| ppObject | [out] An interface pointer to the requested manager, or NULL if an invalid manager was requested. |

6

7     **HResults**

8     **See <u>Common HResults</u>**

9     **E_NOINTERFACE.** The requested interface is not supported.

10

11 **1.12.2. IRuntimeDebugManager**

12     For debugging purposes, hosts 132 may want to group tasks by some host-

13 specific logical construct like a connection, session or request. In this way, a

14 developer who is debugging a particular session (for example) only sees the tasks

15 involved in processing that session – it does not see every task running in the

16 process. This interface provides methods that allow hosts 132 to associate a list of

1  runtime tasks with a given id and friendly name.  In this implementation, the id,

2  the name, and the list of associated tasks have meaning independent of the runtime

3  130 (for purposes of this call).  That is, the runtime 130 blindly passes the

4  parameters on to the debugger.

5

6      The Get and SetDacl methods allow the host 132 to either set or retrieve the

7  ACL's on events and shared memory used by the debugger.  If the events or shared

8  memory are already in use, setting new ACL's will fail.  Likewise, upon creation,

9  restrictive ACL's can be set which disallow debugging (access-denied ACE's in

10  the ACL). If the caller wants to preserve the semantic of the default ACL values

11  when calling SetDacl, the Administrators group and  the current process owner

12  may be added to the ACL, in addition to other required users.  If GetDacl is called

13  before SetDacl, the returned ACL is the default ACL placed on the debug memory

14  block by the runtime 130.
15

16      interface IRuntimeDebugManager: IUnknown

17      {

18      HRESULT BeginConnection(

19          [in] CONNID dwConnectionId,

20          [in, string] wchar_t * szConnectionName);

21      HRESULT SetConnectionTasks(

22          [in] CONNID dwConnectionId,

23          [in] DWORD dwCount,

24          [in, size_is(dwCount)] IRuntimeTask **ppRuntimeTask);

1        HRESULT EndConnection([in] CONNID dwConnectionId);

2        HRESULT SetDacl([in] PACL pacl);

3        HRESULT GetDacl([out] PACL* ppacl);

4        HRESULT IsDebuggerAttached([out] BOOL *pbAttached);

5

6        HRESULT AllowFileLineInfo([in] BOOL fAllowInfo);

7        HRESULT LoadPdb([in] UINT64 id,[in] IStream *pStmPdb);

8     }

9

10    *1.12.2.1.    BeginConnection*

11       IRuntimeDebugManager has three methods for associating tasks with id's

12   and a name. These methods are called in a specific order: BeginConnection are

13   called first to establish a new connection with which to associate tasks. Next,

14   SetConnectionTasks is called to provide the array of runtime tasks for a given

15   connection. Finally, EndConnection is called when the association between the id,

16   name and tasks is no longer needed. Note that although the methods for a given

17   connection are called in order, calls for different connections could be nested. For

18   example, the following call sequence is valid:

19   BeginConnection(1, "C1);

20   SetConnectionTasks(1, ......);

21   BeginConnection(2, "C2");

22   EndConnection(1);

23   SetConnectionTasks(2,.......);

24   EndConnection(2);

25

| Parameter | Description |
|---|---|
| dwConnectionId | [in] An identifier to associate with the list of runtime tasks. This id is completely defined by the host 132 – it is opaque to the runtime. Id's cannot be 0. |
| szConnectionName | [in] A friendly name to associate with the list of runtime tasks. Debuggers may choose to use this name in the user interface to identify the logical grouping of tasks. szConnectionName may not be the NULL string. |

1

2     **HResults**

3     **See <u>Common HResults</u>**

4     **E_INVALIDARG.** dwConnectionId was 0 or BeginConnection was already

5         called with this id, or szConnectionName was null.

6     **E_OUTOFMEMORY.** Memory could not be allocated to hold the list of

7         tasks associated with this connection.

8

9     *1.12.2.2.*     *SetConnectionTasks*

10         After BeginConnection is called, SetConnectionTasks is used to associate a

11     list of runtime tasks with an id and a name. SetConnectionTasks can be called

12     multiple times with the same connection id. However, each subsequent call will

13     overwrite the list made by the previous call. For example, if SetConnectionTasks

14     is called with "T1, T2, T3", then called again with "T2, T3, T4", the list of tasks

15     associated with the id will be "T2, T3", T4".

1

| Parameter | Description |
|---|---|
| dwConnectionId | [in] An id that was previously passed to BeginConnection. |
| dwCount | [in] The number of assemblies in the ppRuntimeTasks array. dwCount cannot be 0. |
| ppRuntimeTasks | [in] An array of IRuntimeTask pointers associated with this id. The array of tasks will contain at least one element. |

2

3  **HResults**

4  **See Common HResults**

5  **E_INVALIDARG.** BeginConnection was not called with dwConnectionId,

6  dwCount or dwConnectionId was 0, the array of runtime tasks didn't

7  contain any elements, or one of the elements was NULL.

8

9  *1.12.2.3.    EndConnection*

10  Disassociates a list of tasks with the given connection id.

11

| Parameter | Description |
|---|---|
| dwConnec | [in] An id that was previously passed to BeginConnection. |

| tionId | |
|---|---|
| | |

1

2      **HResults**

3      **See <u>Common HResults</u>**

4      **E_INVALIDARG.** BeginConnection was not called with dwConnectionId

5      or dwConnectionId was 0.

6

7 *1.12.2.4.*     *SetDacl*

| Parameter | Description |
|---|---|
| pacl | [in] pointer to DACL to be set on events and shared memory |

8

9 *1.12.2.5.*     *GetDacl*

| Parameter | Description |
|---|---|
| pattrs | [out] returns pointer to DACL enforced on events and shared memory. The memory is allocated using CoTaskMemAlloc by the runtime and freed by the host 132 using CoTaskMemFree. If SetDacl has not been called, GetDacl returns the default ACL placed on the debug memory |

| | block by the runtime 130. |
|---|---|

1

## 1.12.2.6. *IsDebuggerAttached*

This method allows hosts 132 to determine if a debugger is attached to the process.

| Parameter | Description |
|---|---|
| pbAttached | [out] returns Boolean true if debugger attached, false if not. |

## 1.12.2.7. *AllowFileLineInfo*

This method allows hosts 132 to control whether file and line info is included in call stacks.

| Parameter | Description |
|---|---|
| fAllowInfo | [in] Host 132 tells runtime 130 whether file and line info may be included in call stacks |

## 1.12.2.8. *LoadPdb*

Used by host 132 to pass module pdb information to runtime for debugging. All modules have substantially unique ids.

1

| Parameter | Description |
|-----------|-------------|
| id | [in] unique id of the module for which to load the pdb |
| pStmPdb | [in] open stream pointing at PDB to be loaded |

2

3 **1.12.3. IRuntimePolicyManager**

4 **<u>Reliability Escalation Policy</u>**

5     The runtime 130 Hosting layer is built to accommodate a variety of hosts,

6 many of which will have different tolerances for handling errors that occur while

7 running managed code in the process. For example, hosts 132 with largely

8 stateless programming models, for example, such as ASP.NET can use a process

9 recycling model to reclaim processes deemed unstable. In contrast, hosts 132 such

10 as SQL Server and WINDOWS shell rely on the process being stable ideally for

11 an infinite amount of time.

12

13     The runtime 130 supports these different reliability needs through an

14 infrastructure that can keep a single AppDomain or an entire process consistent in

15 the face of various situations that would typically compromise stability. Examples

16 of these situations include a thread that gets "stuck" while being aborted (looping

17 finalizer, for example), or the inability to allocate a resource such as memory.

18

1    In general, the runtime may throw an exception on resource failures and

2    thread aborts. However, there are cases where a host 132 may want to override

3    these defaults. For example, consider the case where a failure to allocate memory

4    occurs in a region of code that may be sharing state across threads. Because such

5    a failure may leave the domain in an inconsistent state, the host 132 may choose to

6    unload the entire domain. While this action clearly affects all code running in the

7    domain, it guarantees that the rest of the domains remain consistent and the

8    process remains stable. In contrast, a different host 132 may be willing to allow

9    the "questionable" domain to keep running and instead will stop sending new

10   requests into it and kill the domain itself later.

11

12   Hosts 132 specify which actions to take in these scenarios by calling

13   methods on IRuntimePolicyManager. This interface allows the host 132 to set

14   timeout values for actions like aborting a thread or unloading an AppDomain, and

15   to provide policy statements that govern the behavior when a request for a

16   resource cannot be granted, or when a given timeout expires. All policy

17   statements set through this interface apply to all threads and AppDomains in a

18   process. Policy can not be different for individual threads or domains. The types

19   of operations for which policy can be applied, and the policies themselves are

20   represented by the ERuntimeOperation and EPolicyAction enumerations.

21

22   When setting these policies, it's important to understand exactly what

23   guarantees the runtime 130 makes in terms of cleanup when a thread is aborted, an

24   AppDomain unloaded or the process shut down. These guarantees will be

1 different depending on whether the runtime 130 is terminating managed code

2 gracefully or forcefully.

3

4      Timeouts specified as part of reliability policy are not guaranteed. That is,

5 the runtime 130 cannot guarantee the timeout will occur exactly when specified.

6 Also, when the timer fires, there is no guarantee that the runtime 130 will return

7 immediately.

8

```
9    typedef enum
10   {
11      OPR_ThreadAbort,
12      OPR_ThreadRudeAbortInNonCriticalRegion,
13      OPR_ThreadRudeAbortInCriticalRegion,
14      OPR_AppDomainUnload,
15      OPR_AppDomainRudeUnload,
16      OPR_ProcessExit,
17      OPR_FinalizerRun,
18      MaxClrOperation
19   } EClrOperation;
20
21   typedef enum
22   {
23      FAIL_NonCriticalResource,
24      FAIL_CriticalResource,
25      FAIL_FatalRuntime,
26      FAIL_OrphanedLock,
27      MaxClrFailure
28   } EClrFailure;
29
30   typedef enum
31   {
32      eNoAction,
33      eThrowException,
34      eAbortThread,
35      eRudeAbortThread,
36      eUnloadAppDomain,
37      eRudeUnloadAppDomain,
```

```
1          eExitProcess,
2          eFastExitProcess,
3          eRudeExitProcess,
4          eDisableRuntime,
5          MaxPolicyAction
6        } EPolicyAction;
7
8      interface IRuntimePolicyManager: IUnknown
9        {
10         HRESULT SetDefaultAction(
11             [in] EClrOperation operation,
12             [in] EPolicyAction action);
13
14         HRESULT SetTimeout(
15             [in] EClrOperation operation,
16             [in] DWORD dwMilliseconds);
17
18         HRESULT SetActionOnTimeout(
19             [in] EClrOperation operation,
20             [in] EPolicyAction action);
21
22         HRESULT SetTimeoutAndAction(
23             [in] EClrOperation operation,
24             [in] DWORD dwMilliseconds,
25             [in] EPolicyAction action);
26
27         HRESULT SetActionOnFailure(
28             [in] EClrFailure failure,
29             [in] EPolicyAction action);
30        }
31
```

## 1.12.3.1.    Enum: EClrOperation

The EClrOperation enumeration describes the set of operations for which a host 132 can supply policy actions. A host 132 can associate timeouts with many of these operations as well. Most of the operations are described by their names, but the terms "critical", "non-critical" and "rude" are worth explaining further.

1    The runtime 130 reliability infrastructure distinguishes between aborts or

2    resources failures that occur in "critical" regions of code vs. "non-critical" regions

3    of code. This distinction is made so that hosts 132 can set different policies

4    depending on where in the code the failure occurred. A critical region of code is

5    defined as any place in which the runtime 130 cannot guarantee that the effects of

6    failing a resource request or aborting a thread will affect only the current task. For

7    example, consider the case where a given task is holding a lock and receives a

8    failure when trying to allocate memory. In this scenario, aborting just the current

9    task is not sufficient to ensure stability of the AppDomain because there may be

10   other tasks in the domain waiting on the same lock. If the current task is

11   abandoned, other tasks could be hung forever waiting on the lock. As a result, the

12   host 132 may decide it is better to unload the entire AppDomain instead of running

13   the risk of instability. In contrast, a non-critical region of code is any code in

14   which the runtime 130 can guarantee that an abort or failure will affect only the

15   task on which the error occurs.

16

17   Abort operations of EClrOperation include a notion of a "rude" abort.

18   Generally speaking, a rude abort differs from a "graceful" abort in that a lesser

19   attempt is made to run user backout or cleanup code such as catch blocks, finally

20   blocks, finalizers, and/or the like.

21

22   *1.12.3.2.     Enum: EClrFailure*

23   EClrFailure describes the set of failures a host 132 can set policy actions

24   for. Enum values are:

1          **FAIL_NonCriticalResource.** A failure occurred while trying to

2    allocate a resource in a non-critical region of code (see EClrOperation above

3    for the definition of non-critical region of code). Resources include memory,

4    threads, and locks.

5          **FAIL_CriticalResource.** A failure occurred while trying to allocate a

6    resource in a critical region of code (see EClrOperation above for the definition

7    of critical region of code). Resources include memory, threads, and locks.

8          **FAIL_FatalRuntime.** The runtime 130 is no longer capable of running

9    managed code in the process. All further calls into the host 132ing api will

10    return HOST_E_RuntimeNOTAVAILABLE.

11

12    *1.12.3.3.    Enum: EPolicyAction*

13    This enumeration describes the reliability policies a host 132 can set for the

14    operations defined by EClrOperation and the failures described by EClrFailure.

15    eDefaultAction indicates that the default action for a given operation is acceptable

16    (see SetDefaultAction below). eDisableRuntime disables the runtime 130 in the

17    sense that no more managed code will be run in the process. Once the runtime

18    130 is disabled there is no way to re-enable it. The meanings of the other values in

19    the enumeration follow from their names – abort the thread, rudely abort the

20    thread, gracefully exit the process, and so on.

21

| Value from EPolicyAction | Action taken when condition occurs |
|---|---|
|  |  |

| | |
|---|---|
| eNoAction | No Action is taken |
| eThrowException | Exception is thrown (based on the condition. I.e. stack overflow, out of memory) |
| eAbortThread | Runtime attempts a graceful thread abort. Attempts to run all finally blocks. Catch blocks specific to thread abort, Exception (base class) or catch(...) will also run. |
| eRudeAbortThread | Runtime rudely aborts thread. Catch and finally blocks attributed as 'WillRun' are run. |
| eUnloadAppDomain | Runtime attempts graceful AppDomain unload. Attempt to run finalizers. |
| eRudeUnloadAppDomain | Only 'WillRun' attributed finalizers will be run. All threads with this |

| | |
|---|---|
| | AppDomain in their stack will receive thread abort exceptions, but only 'WillRun' attributed code can catch the exception. |
| eExitProcess | Runtime attempts graceful process exit.<br><br>Finalizers will run and runtime cleanup and logging will occur. |
| eFastExitProcess | Finalizers will not run.<br><br>Internal logging is flushed.<br><br>Notification sent to debugger / profiler. |
| eRudeExitProcess | · Finalizers will not run.<br><br>Some internal runtime logging will not occur. |
| eDisableRuntime | Runtime enters disabled state. No further managed code can be run in this process. Threads will be blocked from entering runtime. |

1   *1.12.3.4.       SetDefaultAction*

2        Hosts 132 can use SetDefaultAction to change how the runtime 130

3   behaves for the operations defined in EClrOperations.  For example, a host 132

4   could use this to always rudely abort threads instead of waiting for them to

5   gracefully stop.  Not all policy actions can be specified as the default behavior for

6   all runtime 130 operations.  Typically, SetDefaultAction can only be used to

7   "escalate" behavior.  For example, a host 132 can not specify that all rude thread

8   aborts be turned into thread aborts instead.  The following table lists the valid

9   policy actions for each operation.

10

| Value from EClrOperation | Valid values from EPolicyAction |
|---|---|
| OPR_ThreadAbort | eAbortThread, eRudeAbortThread, eUnloadAppDomain, eRudeUnloadAppDomain, eExitProcess, eFastExitProcess, eRudeExitProcess, eDisableRuntime |

| | |
|---|---|
| ion     OPR_ThreadRudeAbortInNonCriticalReg<br><br>OPR_ThreadRudeAbortInCriticalRegion | eRudeAbortThread,<br><br>eUnloadAppDomain,<br><br>eRudeUnloadAppDoma<br>in,<br><br>eExitProcess,<br><br>eFastExitProcess,<br><br>eRudeExitProcess,<br><br>eDisableRuntime |
| OPR_AppDomainUnload | eUnloadAppDomain,<br><br>eRudeUnloadAppDoma<br>in,<br><br>eExitProcess,<br><br>eFastExitProcess,<br><br>eRudeExitProcess,<br><br>eDisableRuntime |
| OPR_AppDomainRudeUnload | eRudeUnloadAppDoma<br>in,<br><br>eExitProcess, |

| | |
|---|---|
| | eFastExitProcess, <br><br> eRudeExitProcess, <br><br> eDisableRuntime |
| OPR_ProcessExit | eExitProcess, <br><br> eFastExitProcess, <br><br> eRudeExitProcess, <br><br> eDisableRuntime |
| OPR_FinalizerRun | eNoAction, <br><br> eAbortThread, <br><br> eRudeAbortThread, <br><br> eUnloadAppDomain, <br><br> eRudeUnloadAppDomain, <br><br> eExitProcess, <br><br> eFastExitProcess, <br><br> eRudeExitProcess, <br><br> eDisableRuntime |

1

2     The parameters for SetDefaultAction are:

| Parameter | Description |
|---|---|
| operation | [in] A value from EClrOperation indicating the operation for which you'd like to customize the default behavior. |
| action | [in] A value from EPolicyActions that specifies the action to take when the specified operation occurs. |

3

4     **HResults**

5     **See Common HResults**

6     **E_INVALIDARG.**   An invalid action was passed in for the specified

7     operation.

8

9     *1.12.3.5.     SetTimeout*

10    SetTimeout allows hosts 132 to specify timeout values for thread abort,

11    AppDomain unload and process exit.  By default, the runtime 130 has no timeout

12    value for thread abort and AppDomain unload.  Attempts to abort a thread will

13    continue infinitely until a call to Thread.RudeAbort[1] is called.  Similarly, the

---

1    runtime 130 will continue trying to unload an AppDomain until a call to

2    AppDomain.RudeUnload is received. The default timeout value for process exit is

3    40 seconds, after which the process is rudely shut down.

4

5        If a timeout is specified using SetTimeout, the host 132 will likely also

6    want to specify an action to take on that timeout by calling either

7    SetActionOnTimeout or SetTimeoutAndAction, both of which are documented

8    below.

9

| Parameter | Description |
|-----------|-------------|
| operation | [in] A value from EClrOperation indicating the operation for which you'd like to set a timeout. Valid values are: OPR_ThreadAbortInNonCriticalRegion OPR_ThreadAbortInCriticalRegion OPR_AppDomainUnload OPR_ProcessExit OPR_FinalizerRun (maybe?!) |
| dwMilliseconds | [in] The new timeout value in milliseconds. Passing INFINITE will cause the operation to not time out. |

10

11    **HResults**

1       **See Common HResults**

2       **E_INVALIDARG.** A timeout cannot be set for the specified operation.

3

4       *1.12.3.6.        SetActionOnTimeout*

5       Hosts 132 can specify actions to take when timeout values occur for certain

6       operations. These timeout values may either be the runtime 130 defaults or a host-

7       specified timeout set by calling SetTimeout.

8

9       In this implementation, not all operations, nor all policy actions, are valid

10      when calling SetActionOnTimeout, wherein operations for which timeouts may be

11      supplied are described in the SetTimeout documentation. The policy actions that

12      apply to the operations are those described in table above under SetDefaultAction.

13      For example, the valid actions to perform when an AppDomain unload times out

14      are eRudeUnloadAppDomain, eExitProcess, eRudeExitProcess, eDisableRuntime.

15

| Parameter | Description |
|-----------|-------------|
| operation | [in] A value from EClrOperation indicating the operation for which you'd like to set a timeout. See description of SetTimeout for valid values. |
| action | [in] A value from EPolicyAction specifying the policy action to apply to the operation. See the table under |

| | SetDefaultAction for a description of the valid actions for a given operation. |
|---|---|

1

2    **HResults**

3    **See <u>Common HResults</u>**

4    **E_INVALIDARG.** A timeout cannot be set for the specified operation.

5

6    *1.12.3.7.    SetTimeoutAndAction*

7    SetTimeoutAndAction is a convenience routine that combines the

8    functionality of SetTimeout and SetActionOnTimeout. The default timeouts and

9    valid combinations of operations and policy actions are those described above for

10   these two methods.

11

| Parameter | Description |
|---|---|
| operation | [in] As desribed for SetTimeout |
| dwMilliseconds | [in] As described for SetTimeout |
| action | [in]    As    described    for SetActionOnTimeout |

12

13   **HResults**

14   **See <u>Common HResults</u>**

1    **E_INVALIDARG.**  A timeout cannot be set for the specified operation, or

2    an invalid policy action was specified for the operation.

3

4    *1.12.3.8.      SetActionOnFailure*

5        By default, the runtime 130 throws an exception when it fails to allocate a

6    resource such as memory.   SetActionOnFailure allows the host 132 to override

7    this default and provide a policy action that describes how the runtime 130

8    behaves when a resource cannot be allocated or when a fatal error occurs.

9

| Parameter | Description |
|---|---|
| failure | [in] A value from EClrFailure indicating which type of failure you'd like to set a policy action for. |
| action | [in] A value from EPolicyAction specifying the policy action to apply to the operation.  The valid actions depend on the particular failure.  Valid actions for FAIL_NonCriticalResource and FAIL_CriticalResource are: <br><br> eNoAction, <br><br> eThrowException, <br><br> eAbortThread, |

| | |
|---|---|
| | eRudeAbortThread, <br><br> eUnloadAppDomain, <br><br> eRudeUnloadAppDomain, <br><br> eExitProcess, <br><br> eRudeExitProcess, <br><br> eDisableRuntime. <br><br> The only valid action for FAIL_FatalRuntime is eRudeExitProcess (when a fatal error occurs the runtime 130 is by default disabled). |

1

2      **HResults**

3      **See Common HResults**

4      **E_INVALIDARG.**  A timeout cannot be set for the specified operation, or

5   an invalid policy action was specified for the operation.

6

7   **1.12.4. IRuntimeHostProtectionManager**

8      The .Net Framework class library provides an extensive set of built in

9   functionality to hosted user code.  In addition, numerous third party class libraries

10   exist that provide everything from statistical and math libraries to libraries of new

1    UI controls. Yet, the full extent of functionality provided by a set of class libraries

2    may not be appropriate in particular hosting scenarios. For example, displaying

3    user interface in server programs or services is not useful, or allowing user code to

4    exit the process cannot be allowed in hosts 132 that require long process lifetimes.

5

6    IRuntimeHostProtectionManager provides the host 132 with a means to

7    block classes, methods, properties and fields that offer a particular category of

8    functionality from being loaded in the process. A host 132 may choose to prevent

9    the loading of a class or the calling of a method for a number of reasons including

10    reliability and scalability concerns, or because the functionality doesn't make

11    sense in that host's environment as in the examples described above.

12

13    A host's designation of which categories of functionality to restrict can be

14    thought of as falling into three general buckets:

15    • <u>Categories of functionality that may cause the hosted user code to be</u>

16    <u>instable</u>, but do not affect the overall host 132 stability. These categories

17    may not be callable by partially-trusted code, but are ok to be called from

18    fully-trusted code.

19    • <u>Categories of functionality that do cause host 132 instability (i.e process</u>

20    <u>exit)</u>. In this implementation, members in these categories are not be called

21    in the host 132 process, regardless of the amount of trust the code has been

22    granted.

23    • <u>Categories of API's currently covered by Code Access permissions</u> that a

24    host 132 chooses to deny even in full-trust context. Hosts 132 restrict use

25    of these API's via the *RefusedSet* property of the AppDomainManager

1  class, which effectively injects a CAS deny at the tip of the thread's

2  security evaluation.

3

4  typedef enum
5  {
6    eNoCategory = 0,
7    eSynchronization = 0x1,
8    eSharedState = 0x2,
9    eExternalProcessMgmt = 0x4,
10    eSelfAffectingProcessMgmt = 0x8,
11    eExternalThreading =   0x10,
12    eSelfAffectingThreading = 0x20,
13    eSecurityInfrastructure = 0x40,
14    eUI = 0x80 ,          .
15    eAll = 0xff
16  } EHostProtectionCategories
17

18  interface IRuntimeHostProtectionManager: IUnknown
19  {
20    HRESULT SetProtectedCategories([in] EApiCategories categories);
21    HRESULT SetInaccessibleCategories([in] EApiCategories categories);
22  }
23

### 24  *1.12.4.1.    Enum: EApiCategories*

25     The categories of functionality that a host 132 can choose to "turn off" are

26  described by the EApiCategories enumeration. The categories are:

27  • **eSynchronization**. User code that does its own synchronization can cause

28     scalability issues for server hosts 132 that are turned for high throughput.

29     Methods that allow users code to hold locks (i.e Monitor.Enter()) are good

30     examples of the functionality in the eSychronization category.

31  • **eSharedState**. Sharing state is related to synchronization in the sense that

32     locking primitives are necessary to be sure shared state is kept consistent.

33     Particular hosts 132 may choose to unload the entire AppDomain when a

1    resource failure occurs and a lock is held (see <u>Reliability Escalation</u>

2    <u>Policy</u>).  In these scenarios, it is useful for the host 132 to block members

3    that expose shared state.

4    • **eExternalProcessMgmt**. There are two process management categories.

5    External process management refers to those members that allow new

6    processes to be created, manipulated and destroyed.  Although these actions

7    may affect the scalability of some host 132 environments, they will not

8    affect the stability of the host 132 process itself.

9    • **eSelfAffectingProcessMgmt**. Self-affecting process management refers to

10    those actions that do affect the stability of the host 132 process.  Allow user

11    code to exit the host 132 process is a common example of a member in this

12    category.

13    • **eExternalThreading**. There are also two categories of threading related

14    functionality.  User code that creates or destroys threads can also have an

15    impact on the scalability of a host.  Server hosts 132 may want to have full

16    control over thread lifetime so it can be tracked and tuned for high

17    throughput.

18    • **eSelfAffectingThreading**. Self-affecting threading refers to those actions

19    that do affect the stability of the host 132 process.  Allow user code to abort

20    the current thread is a common example of a member in this category.

21    • **eUI**. An indication that user code that requires human interaction may not

22    be appropriate for server hosts 132 such as web, application or database

23    servers.

24    **eSecurityInfrastructure**. This category refers to functionality provided by

25    the security system itself.  Hosts 132 would block this category of

1     functionality if they want to prevent user code from doing role based

2     security, impersonation and so on.

3     • **eMayLeakOnAbort.** This category refers to methods which may leak

4     memory or resources upon abort.

5

6     *1.12.4.2.*    *SetProtectedCategories*

7     Hosts 132 use SetProtectedCategories to specify which categories of

8     functionality may NOT be available to partially trusted code. The default (i.e., the

9     behavior if this method is not called) is that all members are allowed.

10

| Parameter | Description |
|-----------|-------------|
| Categories | [in] Values from EHostProtectionCategories indicating which categories to block. |

11

12     **HResults**

13     **See <u>Common HResults</u>**

14

15     *1.12.4.3.*    *SetInaccessibleCategories*

16     Hosts 132 use SetInaccessibleCategories to specify which categories of

17     functionality may NOT be called in the host 132 process – not even by fully

18     trusted code. The default (i.e the behavior if this method is not called) is that all

19     members are allowed.

1

| Parameter | Description |
|-----------|-------------|
| Categories | [in]  Values  from EHostProtectionCategories  indicating which categories to block. |

2

3      **HResults**

4      **See Common HResults**

5

6      *1.12.4.4.      SetEagerSerializeGrantSets*

7      This method may be called once before starting the runtime to sacrifice a

8      little assembly-loading-performance for a guarantee that a certain rare race

9      condition that can result in a FatalEE error won't occur.

10

11      *1.12.4.5.      Enum EClrEvent*

12      These values enumerate the events for which hosts 132 can register

13      callbacks.  Registering for runtime events is described below in the section on

14      IRuntimeOnEventManager.

15

16          typedef enum
17          {
18             Event_DomainUnload,
19             Event_ClrDisabled,
20             MaxClrEvent
21          } EClrEvent;
22

1 **1.12.5. IRuntimeOnEventManager**

2    Hosts 132 may register callbacks with the runtime to be informed of

3 important internal events.  Currently there are two defined events, 'appdomain

4 unload' and 'clr disabled' which indicates a fatal error.  Hosts 132 register for

5 these events by first retrieving the IRuntimeOnEventManager interface from the

6 IRuntimeControl.

7

8    NOTE: These events may be fired more than once and from different

9 threads to signal an appdomain unload or the disabling of the runtime 130.

10

11        interface IRuntimeOnEventManager: IUnknown

12        {

13            HRESULT RegisterActionOnEvent(

14                [in] EClrEvent        event,

15                [in] IActionOnRuntimeEvent *pAction

16                );

17            HRESULT UnregisterActionOnEvent(

18                [in] EClrEvent        event,

19                [in] IActionOnRuntimeEvent *pAction

20                );

21        }

22 *1.12.5.1.    RegisterActionOnEvent*

| Parameter | Description |
|-----------|-------------|
|           |             |

| event | [in] Values from EClrEvent indicating for which event this callback is registered. |
| pAction | [in] pointer to host 132 callback. The runtime will call this when the registered action occurs. |

1

2 *1.12.5.2.* ***UnregisterActionOnEvent***

| Parameter | Description |
|---|---|
| event | [in] Values from EClrEvent indicating for which event this callback is registered. |
| pAction | [in] pointer to host 132 callback. The runtime will call this when the registered action occurs. |

3

4 **1.12.6. IActionOnRuntimeEvent**

5

```
6        interface IActionOnRuntimeEvent: IUnknown
7        {
8          HRESULT OnEvent(
9              [in] EClrEvent   event,
10             [in] IUnknown    *data
11             );
12        }
```

13

1  *1.12.6.1.    OnEvent*

| Parameter | Description |
|---|---|
| Event | [in] Values from EClrEvent indicating for which event this callback is registered. |
| Data | [in] If the event is Event_DomainUnload, data is a pointer to that AppDomain. In other cases, data is NULL. |

2

3  **1.12.7. IRuntimeGCManager**

4
5      interface IRuntimeGCManager : IUnknown
6      {
7          HRESULT Collect([in] LONG Generation);
8          HRESULT GetStats([in][out] COR_GC_STATS *pStats);
9          HRESULT SetGCStartupLimits([in] DWORD SegmentSize, [in]
10    DWORD
11         MaxGen0Size);
12         }

13  *1.12.7.1.    Collect*

14      Forces a collection to occur for the given generation, regardless of current

15  GC statistics.

16

| Parameter | Description |
|---|---|
| Generation | [in] Generation to collect. In this |

| | |
|---|---|
| | implementation, a value of -1 means collect all generations. |

1

## 1.12.7.2. GetStats

3     ·      GetStats returns a set of current statistics about the state of the GC system.

4 These values can then be used by a smart allocation system to help the GC by

5 adding more memory or forcing a collection.

6

| Parameter | Description |
|---|---|
| pStats | [in][out] returns current statistics in COR_GC_STATS structure (described below) |

7

8       This structure is used to return statics for the GC system. Set the Flags

9 value to a bitmask of values that may be returned. Only those values which are

10 requested are calculated and returned to the caller.
11
12       typedef struct _COR_GC_STATS
13
14       {
15      ULONG      Flags;                  // The values to get.
16
17      // Value when COR_GC_COUNTS is specified.
18      SIZE_T     ExplicitGCCount;         // How many times was
19 GC
20      forced to run by external request.
21      SIZE_T     GenCollectionsTaken[3];      // Number of
22 collections
23      done for each generation

```
1
2              // Memory sizes, valid for COR_GC_MEMORYUSAGE.
3              SIZE_T    CommittedKBytes;              // Total committed
4       bytes from
5           all heaps.
6              SIZE_T    ReservedKBytes;               // Total reserved bytes
7       from all
8           heaps.
9              SIZE_T    Gen0HeapSizeKBytes;           // Size of gen 0 heap.
10             SIZE_T    Gen1HeapSizeKBytes;           // Size of gen 1 heap.
11             SIZE_T    Gen2HeapSizeKBytes;           // Size of gen 2 heap.
12             SIZE_T    LargeObjectHeapSizeKBytes;    // Size of large
13      object
14          heap.
15             SIZE_T    KBytesPromotedFromGen0;       // How many
16      bytes
17          promoted to next generation.
18             SIZE_T    KBytesPromotedFromGen1;
19
20          } COR_GC_STATS.
21
```

22 *1.12.7.3.    SetGCStartupLimits*

23     Sets the segment size and gen 0 maximum size.  In this implementation,

24 this value is specified once, and will not change if called later.

25

| Parameter | Description |
|-----------|-------------|
| SegmentSize | [in] Sets segment size. |
| MaxGen0Size | [in] Sets size of Gen 0 |

26

27